

精通 Linux C编程

程国钢　编著

清华大学出版社

精通 Linux C 编程

程国钢　　编著

清华大学出版社
北　京

内 容 简 介

本书深入浅出、循序渐进地讲解了Linux平台下的C程序设计，并通过大量的程序实例，以及综合开发案例的演示，帮助读者快速掌握Linux平台下C语言编程的方法和技巧。

本书内容翔实，共分为三大部分。第一部分为基础篇，共有5章，主要讲解了Linux系统基础、C语言编程基础、Linux下的两种常用文本编辑器vim和Emacs、程序编译器gcc和调试器gdb，以及make管理工具。第二部分为提高篇，共有8章，主要讲解了Linux系统下各种操作的系统调用，包括文件操作、进程控制、进程间通信、线程控制、网络编程、GTK+图形界面编程等。第三部分为实战篇，共有5章，分别为5个不同的Linux平台下C程序开发的综合案例，向读者详细阐述了Linux文件操作、GTK+图形界面编程、Linux网络编程，以及基于Linux平台的嵌入式软件开发的方法和技巧。

本书由作者根据多年来的开发工作经验编著而成，语言通俗易懂，内容丰富，注重实例讲解，知识涵盖面广。非常适合Linux平台下C语言编程的初学者以及高校本科生、研究生阅读，也适合在Linux系统下进行C程序开发的工程师查阅和学习。

图书在版编目(CIP)数据

精通Linux C编程 / 程国钢　编著. —北京：清华大学出版社，2015
ISBN 978-7-302-39367-2

Ⅰ. ①精…　Ⅱ. ①程…　Ⅲ. ①Linux操作系统—程序设计　②C语言—程序设计　Ⅳ. ①TP316.89 ②TP312

中国版本图书馆CIP数据核字(2015)第031610号

责任编辑：刘金喜　蔡　娟
装帧设计：牛静敏
责任校对：成凤进
责任印制：

出版发行：清华大学出版社
　　网　　址：http://www.tup.com.cn，http://www.wqbook.com
　　地　　址：北京清华大学学研大厦A座　　**邮　　编：**100084
　　社 总 机：010-62770175　　**邮　　购：**010-62786544
　　投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
　　质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张：**36.25　　**字　　数：**905千字
(附光盘1张)
版　　次：2015年6月第1版　　**印　　次：**2015年6月第1次印刷
印　　数：1～2500
定　　价：68.00元

产品编号：

前言

PREFACE

Linux 诞生于 1991 年，由 UNIX 发展而来。几乎每天 Linux 都会以某种方式出现在我们的生活中，我们已经数不清在 Linux 上有多少应用程序，以及有多少机构在使用 Linux。国内外无数大型企业都在使用 Linux 系统作为服务器解决方案，尤其是在嵌入式开发领域，Linux 的应用更是在不断增加。在百度上搜索“Linux 软件工程师”的相关网页约有 1 230 000 篇，由此可见，在 Linux 平台下进行程序开发的需求之大。

作者根据自己多年来在 Linux 平台下进行 C 程序开发所积累的经验，并融合大量的程序实例而著成此书。本书由浅入深，适合各个水平阶段的读者学习。

本书特点

1．配套视频讲解光盘

为了让读者更加快速、直观地学习本书内容，作者专门为本书录制了全程多媒体视频教学，包括各个技术知识点，以及程序案例的分析。结合视频的讲解，能够帮助读者更高效地掌握 Linux 下 C 语言编程的技巧与方法。

光盘具体内容有：

- Linux 常用工具使用视频
- Linux 入门视频
- 案例源文件
- 本书全程多媒体教学视频
- 实战演练参考答案

2．循序渐进，由浅入深

本书从 Linux 系统的安装、C 语言编程基础、Linux 下的基本编辑器、程序编译器和调试器、make 工具管理器，到 Linux 系统的各种函数调用，再到 Linux 下具体程序案例的设计开发，内容由浅入深，囊括了 Linux 下 C 程序开发的各个环节。

3．程序实例丰富，实践性强

在本书中，几乎每个知识点都会伴随一个或多个程序实例，通过实例来加深和巩固读者对知识点的理解和掌握。对于每个程序实例，作者都添加了十分详细的注释，方便读者理解。并且，对于所有的实例，读者都可以在自己的实验环境中完整实现。尤其是第 3 部分的 5 个项目

案例，更是完整地向读者演示了 Linux 环境下项目实例的设计与开发。

4．技术全面，知识点阐述到位

网络编程和图形界面编程是 C 程序学习中比较深入的知识，也是本书重点讲解的内容。在第 3 部分，我们将这些知识点与实际的项目开发结合，通过逐步设计与实现，深化和巩固读者对它们的理解与掌握。将 Linux 系统开发技术、C 语言开发技术、软件工程思想融会贯通，使得本书成为思想和内容都极其丰富的图书。

本书内容

第 1 章：介绍 Linux 系统的基本概念和安装方法，Linux 下的常用命令，以及 Shell 的使用。这些是使用 Linux 的基础，帮助读者为本书后续的学习打下扎实的基础。

第 2 章：详细讲述了 C 语言的编程基础，它们是熟悉和掌握 C 语言的必备知识，同时也方便了读者在阅读本书时查阅 C 语言中的相关知识点。

第 3 章：讲述 Linux 下最常用的两种文本编辑器 vi 和 Emacs，并通过实例讲解让读者一步步地学会如何使用这些编辑器。

第 4 章：讲解 Linux 系统下的程序编译器 gcc 和程序调试器 gdb，两者是在 Linux 下进行 C 程序开发所必备的工具。

第 5 章：讲述 Linux 下的工程管理器 make，以及 Makefile 的书写规则。make 工具大大提高了实际项目的开发效率，几乎所有 Linux 下的项目编程都会涉及它。

第 6 章：讲解基于文件描述符的文件 I/O 操作，以及 Linux 中文件系统的概念。文件操作是 Linux 系统中最常见的操作之一，在 Linux 中，所有的内容都被看成文件，所有的操作都可以归结为对文件的操作。

第 7 章：讲述基于流的文件 I/O 操作。基于流的 I/O 操作是由标准 C 函数库提供的，与基于文件描述符的 I/O 操作相比，基于流的 I/O 更简单、方便。在大多数情况下，程序员更愿意使用基于流的输入输出方法。

第 8 章：详细阐述了 Linux 下进程控制的原理。进程是操作系统中一个非常重要的概念，熟悉使用进程的操作和进程控制的相关系统调用，会使用户在使用 Linux 系统完成各种工作时更加得心应手。

第 9 章：讲解 Linux 信号的基本概念，以及 Linux 下信号处理的机制。信号的使用对于灵活使用 C 语言在 Linux 环境下进行程序开发是非常有益的，在编写大型的程序时，经常会需要处理多个进程之间的异步事件，所以是离不开信号的使用的。

第 10 章：详细讲述了 Linux 进程间通信的概念，以及 Linux 下 5 种最常用的进程间通信方式，它们包括管道、命名管道、消息队列、共享内存、信号量等。

第 11 章：介绍了 Linux 的线程相关基础知识，如何对其进行操作以及线程间同步的方法。随着多核心处理器的普及，合理使用多线程可以大大提高程序代码的执行效率。

第 12 章：详细讲述了 Linux 网络编程的原理与方法，通过大量的程序实例演示了 Socket

编程中常用 API 的使用方法。

第 13 章：讲述 GTK+图形界面编程。GTK+是 Linux 下基于 C 的图形界面开发库，本章通过程序实例向读者演示了使用 GTK+库创建各种界面元件的方法。

第 14 章：通过讲述一个计算器软件的设计案例，进一步向读者演示图形界面编程的技巧，以及 GTK+信号与回调函数的原理。

第 15 章：通过设计一个类似于 QQ 的聊天软件，向读者演示了小型项目工程软件的模块划分方法，以及 Linux 下的 C 程序开发的步骤。使读者能够更深层次地掌握 GTK+图形界面编程，以及 Linux 网络编程的原理方法。

第 16 章：讲述 Linux 下一种基于 C/S 模式实现的远程管理工具的设计。使读者对 Linux 下的文件 I/O 操作、相关系统服务的管理有更深层次的了解和认识，并进一步掌握 GTK+图形界面编程，套接字 Socket 网络编程的使用方法。

第 17 章：讲述了在 Linux-2.4.20-8 内核下利用 Netfilter 数据控制过滤机制完成简易防火墙软件的设计，实现了对固定端口、网页访问及不同协议类型的数据报文的管理和控制。

第 18 章：讲述基于 Linux 的嵌入式家庭网关远程交互操作平台的设计。本章向读者阐述了嵌入式系统，以及家庭网关的概念。案例采用 B/S 结构的开发模式，嵌入式 Web 服务器选取 Boa，并结合 CGI 技术，实现了动态的具体智能设备的访问和控制。

本书适合的读者：

- Linux 及 C 程序设计初学者。
- 大中专院校计算机及相关专业的学生。
- 高校计算机及相关专业本科生、研究生。
- Linux 软件开发从业人员。
- 开源软件开发爱好者。
- 社会相关培训学员。

最后，我要感谢我的家人及好友陈曦在我最困难的时候带给我快乐与动力，支持我一直坚持下来，并最终完成这部著作。

本书主要由程国钢主持编写，参加本书编写工作的还有许小荣、张泽、刘荣、张璐、王统、周艳丽、刘波、苏静、王冬、王龙、陈作聪、王松年、卿前华、蔡娜、肖岳平、聂阳、沈毅、张华杰等，在此，编者对以上人员致以诚挚的谢意！

作者力图使本书案例功能翔实，并尽量使用关键编程技术进行程序设计和简化程序代码，但由于水平有限，书中难免有错误、纰漏之处，欢迎广大读者、同仁批评斧正。

服务邮箱：wkservice@vip.163.com

编者

2014 年 4 月



目录

CONTENTS

第 I 部分 基础篇

第 I 部分

基础篇



第 1 章

Linux系统概述

Linux 是一套免费使用和自由传播的类 UNIX 操作系统，已发展成为现今世界上最流行的操作系统之一。几乎每天 Linux 都以某种方式出现在媒体上，我们已经数不清在 Linux 上有多少应用程序，以及有多少机构在使用 Linux。本章将简单介绍 Linux 操作系统的基础知识，以帮助读者对所要学习的知识建立起清晰的认识，为以后的学习打下扎实的基础。



本章内容：

◎ 什么是 Linux。
◎ Linux 系统特点及主要功能。
◎ Linux 的内核版本和发行版本。
◎ Linux 系统的安装。
◎ Shell 的使用。
◎ Linux 下的常用命令。

1.1 什么是 Linux

Linux 是一种免费的，提供源代码的，能适用于 PC 机的类似于 UNIX 的网络操作系统，它主要用于基于 Intel x86 系列 CPU 的计算机上。这个系统是由世界各地成千上万的程序员设计和实现的，其目的是建立不受任何商品化软件的版权制约的、全世界都能自由使用的 UNIX 兼容产品。

Linux 操作系统是由 UNIX 发展而来，1969 年由 Ken Thompson 和 Dennis Ritchie 在美国贝尔实验室开发的一种操作系统。由于其良好而稳定的性能迅速在计算机中得到广泛的应用，在随后几十年中也有不断的改进。

UNIX 操作系统正式发布于 1974 年美国计算机学会的杂志 ACM 上，到 1975 年引入了多项技术，从而使它成为一个真正的多用户分时操作系统。此后短短两年时间，又出现了 Xenix、SUNOS 等 UNIX 操作系统的不同版本。1985 年美国麻省理工学院在已有的基础上开发出了 UNIX 操作系统的图形化界面 X Window 系统，它已经成为工作站图形界面的标准。

在 20 世纪 80 年代，Andrew S.Tanenbaum 为了满足教学的需要编写了一个与 UNIX 类似的 Minix 系统。1990 年，芬兰人 Linus Torvalds 接触了 Minix 系统后，开始着手研究编写一个开放的与 Minix 系统兼容的操作系统。1991 年 10 月 5 日，Linus Torvalds 在赫尔辛基技术大学的一台 FTP 服务器上发布了一个消息，这也标志着 Linux 系统的诞生。

1984 年，自由软件的积极提倡者 Richard Stallman 组织开发了一个完全基于自由软件的软件体系 GNU，并拟定了一份通用公共许可证(General Public License，GPL)。GPL 的内容主要是保持软件的免费使用和传播，要求必须以源代码的形式发布软件，并且任何使用者都可以以源代码的形式复制或传播软件给任何人。Linus Torvalds 在 1993 年将 Linux 系统转向 GPL，并加入了 GNU。从而最终使自由软件有了发展根基，即基于 Linux 系统的 GNU。这一版权除了规定有自由软件的各项许可权外，还允许用户出售自己的程序拷贝。

2005 年，此时的 Linux 已经发展到了 2.4 版本，该版本的 Linux 内核提供了对 USB、PC 卡、ISA、蓝牙、RAID 和 EXT3 文件系统等的支持；2.6 版本的 Linux 内核则进一步提供了对 PAE、64 位处理器、16TB 大容量存储器以及 EXT4 文件系统等的支持。

截止到 2014 年，在这段时间内 Linux 的发行版呈现了爆炸式的增强，桌面环境 KDE 发布了 KDE 4.2 版，而桌面环境 GNOME 则发布了 GNOME 3 版本，一个全新的桌面环境 Unity 在 Ubuntu 的 11.04 发行版上出现，而 Linux 内核也发布到了 3.13.3 版(2014 年 2 月 13 日)。在这几年中的另外一个突破则是 Linux 被大量地移植到基于 ARM 等处理器的嵌入式系统中，而基于 Linux 内核的移动端商用操作系统 Android(安卓)也在 2009 年 9 月发布。

绝大多数基于 Linux 内核的操作系统使用了大量的 GNU 软件，包括 shell 程序、工具、程序库、编译器及工具，还有许多其他程序，例如 Emacs。正因为如此，GNU 计划的开创者 Richard Stallman 博士提议将 Linux 操作系统改名为 GNU/Linux。但有些人只把操作系统叫作“Linux”。

1.2 Linux 系统特点及主要功能

Linux 系统是真正的多用户、多任务、多平台的操作系统，提供具有内置安全措施的分层文件系统，支持多达 32 种文件系统。Linux 系统的源代码是开放的，任何人都能修改和重新发布它。另外，Linux 系统提供了强大的管理功能。

1.2.1 Linux 系统特点

Linux 源于 UNIX，从一开始就继承了 UNIX 的先进性，是一个真正的多任务、多用户、具有复杂内核的操作系统。它充分利用了现行的 CPU 的任务切换功能，创造了多任务、多用户环境，允许多个用户同时使用一台计算机系统。同时，多个用户能从相同或不同的终端上用同一个应用程序的副本进行工作，真正实现了多用户的并行操作。与以往操作系统的不同之处在于，它采用抢先式多任务机制，保证每一个程序都有机会运行，每个程序一直执行到操作系统抢占 CPU 让其他程序执行为止，这种机制让 CPU 的功能发挥出最大的作用。

Linux 系统是单内核，这种内核比微内核复杂。在这种内核中，大量的功能是放在内核中直接实现的，而在微内核系统中，许多功能是采用服务进程的形式放在内核外实现的。

Linux 支持现有的常见文件系统。如 Linux ext2、Linux ext3、Linux ext4、FAT16、FAT32、ISO9660 光盘文件系统和 Windows NT 的 NTFS 文件系统等。它具有严谨的文件及目录结构。文件都是按照作用或者性质来存放的。其目录结构是标准的树状结构。此外，Linux 将设备都当成文件来处理。这样，当要使用某一设备时，只需要简单读写该设备文件就行，极大地方便了设备的使用。

Linux 完全支持 POSIX(可移植性操作系统)规范，可以很容易地将 UNIX 下的应用程序移植到 Linux 下。可移植性使 Linux/UNIX 与其他任何机器进行通信成为可能，而不需要增加通信接口。

Linux 系统具有很强的适应性。Windows 操作系统只能运行在 x86 结构的处理器上，各厂商的 UNIX 只能运行在各自的处理器上，但是 Linux 系统几乎能运行在包括 ARM 处理器的所有常见的处理器上。Linux 还支持广泛的外部设备，在 Linux 中可以找到几乎所有的设备驱动程序。

Linux 平台下有大量的应用软件，如电子表格、字处理、数据库、联网工具及游戏等。此外，Linux 可以使用软件包管理系统来安装软件，用 rpm、apt 等命令可以很方便地安装、查询、卸载软件。Linux 还支持一系列的开发工具，几乎所有的主流程序设计语言都可以移植到 Linux 上。

归结起来，Linux 操作系统主要具有以下特点：

- 开放性。
- 多任务和多用户。
- 支持多种硬件平台。
- 可靠的系统，安全、稳定，可用于关键任务。
- X Windows 的 GUI 环境。

- 强大的网络功能。
- 设备独立。
- 支持多种文件系统。
- 置于 GPL(General Public License，共用许可证)保护下，完全免费，可获得源代码，用户可以随意修改它。

1.2.2 Linux 系统的主要功能

Linux 系统为用户提供了强大的管理功能，主要包括存储管理、系统用户和用户组管理、进程管理、文件管理等。

1. 存储管理

Linux 内核采用虚拟页式存储管理，采用三级映射机制实现从线性地址到物理地址的映射。这三级映射机制包括：页面目录(PGD)、中间目录(PMD)和页面表(PT)。具体的映射过程可描述如下：

(1) 从内存的 CR3 寄存器中找到 PGD 基地址。

(2) 以线性地址的最高位段为下标，在 PGD 中找到指向 PMD 的指针。

(3) 以线性地址的次位段为下标，在 PMD 中找到指向 PT 的指针。

(4) 同理，在 PT 中找到指向页面的指针。

(5) 线性地址的最后位段为在此页中的偏移量，这样就完成了从线性地址到物理地址的映射过程。

对于 32 位的微机平台，如 Intel 的 x86 采用段、页式的两层映射机制，而 64 位的微处理器采用三级分页技术。所以对于传统的 32 位平台，Linux 采用让 PMD(中间目录)全 0 的方式来消除中间目录域，这样就把 Linux 逻辑上的三层映射模型落实到 x86 结构物理上的二层映射，从而保证了 Linux 对多种硬件平台的支持。

2. 用户和用户组管理

Linux 系统是一个多用户的操作系统，任何一个要使用系统资源的用户，都必须首先向系统管理员申请一个账号，然后以这个账号的身份进入系统。用户的账号一方面可以帮助系统管理员对使用系统的用户进行跟踪，并控制他们对系统资源的访问；另一方面也可以帮助用户组织文件，并为用户提供安全性保护。每个用户账号都拥有一个唯一的用户名和用户口令。用户在登录时输入正确的用户名和口令后，就能够进入系统和自己的主目录。

实现 Linux 用户账号的管理，要完成的工作主要有以下几个方面：

- 用户账号的添加、删除与修改。
- 用户口令的管理。
- 用户组的管理。

3. 进程管理

Linux 是一个多用户、多任务的分时操作系统。多用户是指多个用户可以在同一时间使用计算机系统；多任务是指 Linux 可以同时执行多个任务，它可以在还未执行完一个任务时又执

行另一项任务。

操作系统管理着多个用户的请求和多个任务的执行。我们知道，大多数系统都只有一个CPU 和一个主存，但可能有多个二级存储磁盘和多个输入/输出设备。

操作系统管理这些资源，并在多个用户间共享资源，当某一个用户提出一个请求时，好像系统只被该用户独自占用。而实际上操作系统监控着一个等待执行的任务队列，这些任务包括用户作业、操作系统任务、系统中断等。

操作系统根据每个任务的优先级别，为它们分配合适的时间片段，每个时间片段大约有零点几毫秒，这足够计算机完成成千上万的指令。每个任务都会被系统运行一段时间，然后挂起，系统转而去处理其他任务；过一段时间以后再回来处理这个任务，直到某个任务完成，才从任务队列中删除。

在 Linux 操作系统中，任务被称为“进程”。进程在其一生中存在多个状态，正是操作系统管理着进程在多个状态之间的轮换，并控制着多个进程之间协调有序地执行。

4. 文件管理

Linux 的文件类型包括文本文件、二进制文件、目录文件、链接文件、设备文件和管道文件等。

Linux 最重要的特征之一就是支持多种不同类型的文件系统。为了支持多种文件系统，Linux 用一个被称为虚拟文件系统(VFS)的接口将真正的文件系统同操作系统及其服务器分离开来，它能掩盖不同文件系统之间的差异，使所有的文件系统在操作系统和用户程序看起来都是相同的。可以说，在 Linux 中，用户所有的操作都可以看作是对操作系统中文件的操作。

由于 Linux 是一个多用户的操作系统，这就要考虑安全性的问题。文件权限是操作系统安全性的一个重要因素。Linux 文件的权限有 3 种：读、写和执行。读权限允许用户查看文件的内容，对目录文件来说，允许用户列出目录中的内容；写权限允许用户写和修改文件，对目录文件来说，写权限允许用户在这个目录中创建新的文件或删除文件；执行权限允许用户运行文件，对目录来说，执行权限允许用户进入和退出该目录。

另外，在 Linux 中，一个分离的文件系统不是通过设备标识(驱动器号)来访问的，而是把它合并到一个单一的目录树结构中，通过目录来实现访问。

1.3 Linux 的内核版本和发行版本

在 Linux 操作系统不断发展壮大的同时，Linux 的内核也在迅速地更新着。Linux 内核的官方版本是由 Linus Torvalds 本人维护着。其内核的版本号形式为 major.minor.patchlevel。patchlevel 是对当前内核版本的修订次数。例如，kernel 2.0.30 表示对内核版本的第 30 次修订。根据约定，次版本号为偶数时表示该内核为稳定发布版本，对它的修订主要是消除各种错误，为其添加新特性；次版本号为奇数时，则表示其为不稳定的开发版本，开发人员在其中添加了新特性。

在普通用户 PC 机上的 Linux 主要是作为 Linux 发行版的一部分而使用。这些发行版由个人、松散组织的团队，以及商业机构和志愿者组织编写。它们通常包括其他的系统软件和应用

软件，以及一个用来简化系统初始安装的安装工具和让软件安装升级的集成管理器。大多数系统还包括了像提供 GUI 界面的 XFree86 之类的曾经运行于 BSD 的程序。发行版为许多不同的目的而制作，包括对不同计算机结构的支持，对一个具体区域或语言的本地化，实时应用，嵌入式系统，甚至许多版本故意只加入免费软件。

目前，超过三百个发行版被积极地开发，最普遍被使用的发行版大约有十几个。一个典型的 Linux 发行版包括 Linux 内核、一些 GNU 程序库和工具、命令行 shell、图形界面的 X Window 系统和相应的桌面环境，如 KDE 或 GNOME，并包含数千种从办公套件、编译器、文本编辑器到科学工具的应用软件。

主流的 Linux 发行版如表 1.1 所示。

表 1.1　Linux 操作系统的发行版

原 始 版 本	最新发行版		
Debian	Debian	Ubuntu	Linux Mint
	Knoppix	MEPIS	sidux
	CrunchBang Linux	Chromium OS	Google chrome OS
Red Hat	Red Hat Enterprise Linux	Fedora	CentOS
	Scientific Linux	Oracle Linux	
Mandriva	Mandriva Linux	PCLinux OS	Unity Linux
	Mageia		
Gentoo	Gentoo Linux	Sabayon Linux	Calculate Linux
	Funtoo Linux		
Slackware	Slackware	Zenwalk	VectorLinux
其他	SUSE	Arch Linux	Puppy Linux
	Damn Small Linux	MeeGo	Slitaz
	Tizen	StartOS	

1.4 系统的安装

学习 Linux，最好的方式莫过于动手与实践，本节将针对 Ubuntu 12.04LTS 版本，带领读者在 PC 机上实现 Linux 操作系统的各种安装方式。

1.4.1　系统安装前的准备工作

在开始安装之前，最好先对系统有一些了解，可以使用计算机说明书，或者在已经安装了 Windows 系统的计算机上选择“我的电脑”→“属性”命令，从里面了解一些相关的计算机配置信息。本节向读者讲解在安装 Linux 操作系统时需要注意的几个问题。

1. 操作系统的安装顺序

安装 Linux 前首先要考虑计算机内是否已安装其他的操作系统，是否要让 Linux 与原有的操作系统并存。若让 Windows 系列操作系统与 Linux 操作系统并存，务必先安装 Windows 系统，再安装 Linux。因为 Windows 系统的安装程序会更改主引导记录，且无法进行各种作业系统的多重启动。假如硬盘上的分区全部都采用 NTFS 文件系统，并且不打算删除其中任何分区上的资料，那就必须建立一个 FAT16 或 FAT32 的分区或再准备一个硬盘，因为 Linux 不支持 NTFS 的写入功能，通过 DOS 格式的分区就能够实现 NTFS 文件系统与 Linux 间的信息传输。

注 意

比较新的 Linux 发行版(例如 Ubuntu 12.04)通常会支持使用某些引导工具来对分区进行引导修改，在这种情况下就不一定需要先安装 Windows。

2. 硬件环境

由于设计 Linux 时的初衷之一就是用较低的系统配置提供高效率的系统服务，所以安装 Linux 并没有严格的系统配置要求，只要 Pentium 以上的 CPU、64MB 以上的内存、1GB 左右的硬盘空间，就能安装基本的 Linux 系统并且能运行各种系统服务。但是随着 Linux 系统的桌面环境的发展，尤其是各种显示效果的扩展，例如 Ubuntu 的 unity 等环境对硬件环境的要求也会越来越高，通常来说推荐使用 2GB 以上的内存，10GB 以上的硬件空间。

3. 网络配置

若要连接局域网，需要先查明 IP 地址、子网掩码、域名服务器、网关等相关设定。了解网卡型号、查看网卡是否被支持等。

4. 外设型号

常用外设的型号要了解。比如鼠标、键盘类型、显卡的型号及各项参数等。

1.4.2 安装 Linux

Linux 系统最简单、方便的安装方法是从 CD-ROM 或者 U 盘安装，用户可以享受最人性化的，类似于 Windows 系列的安装界面。只要将计算机设置成光驱/U 盘引导，把安装光盘放入光驱或者把支持启动的 U 盘插入计算机的 USB 接口，重新引导系统，便会出现安装启动界面，如图 1.1 所示。



图 1.1 Ubuntu 安装欢迎界面

随着 Linux 的发展，和很多年前不一样，现在最新的发行版安装非常简洁方便，基本上只需要进行简单的设置即可完成。

Ubuntu 12.04 在显示安装欢迎界面的同时也会对系统的硬件进行初始化，当初始化完成之后会出现语言和安装设置选项，如图 1.2 所示。

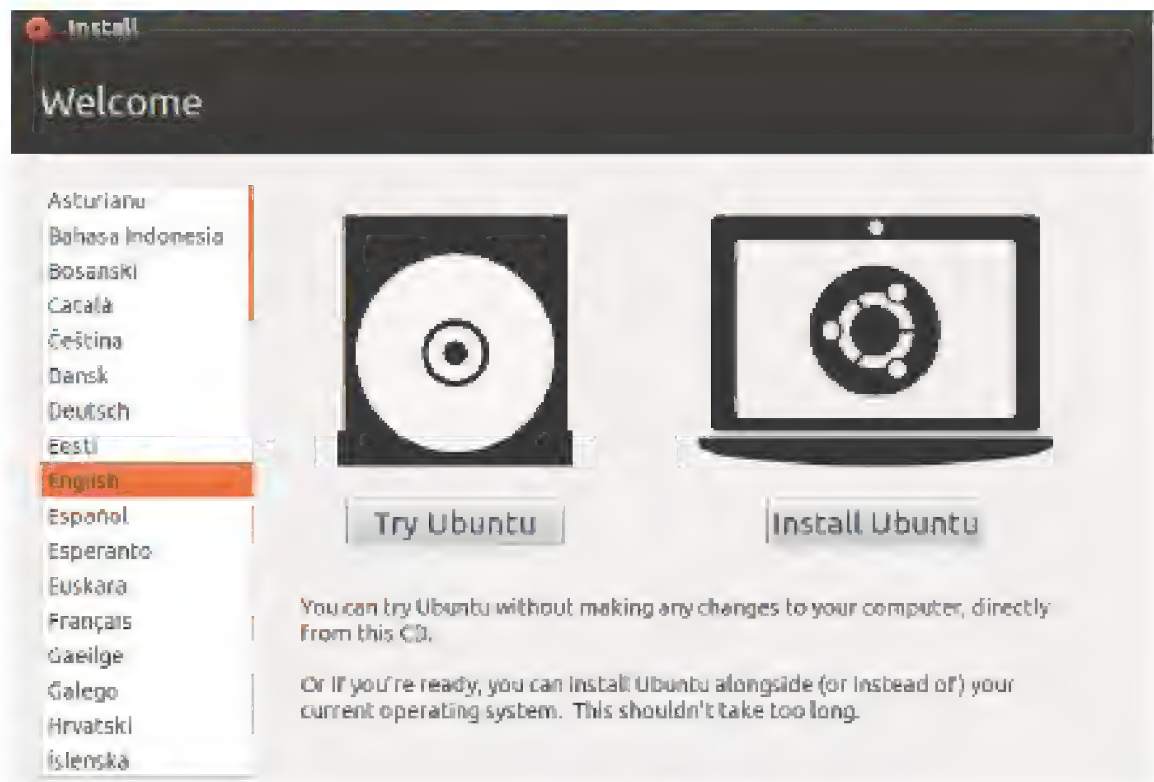


图 1.2　语言和安装设置选项

在语言和安装设置选项中应该拖动左侧的滑动条到下方将操作系统的语言设置为中文(简体)，此时可以看到界面显示如图 1.3 所示，都变成了中文；在右侧提供了“试用 Ubuntu”和“安装 Ubuntu”两个选项，前者是在不安装操作系统的情况下直接使用 Ubuntu 的功能，其提供了包括图形界面等在内的几乎全部功能，但是也有一些限制，后者则是在计算机上安装 Ubuntu 操作系统。

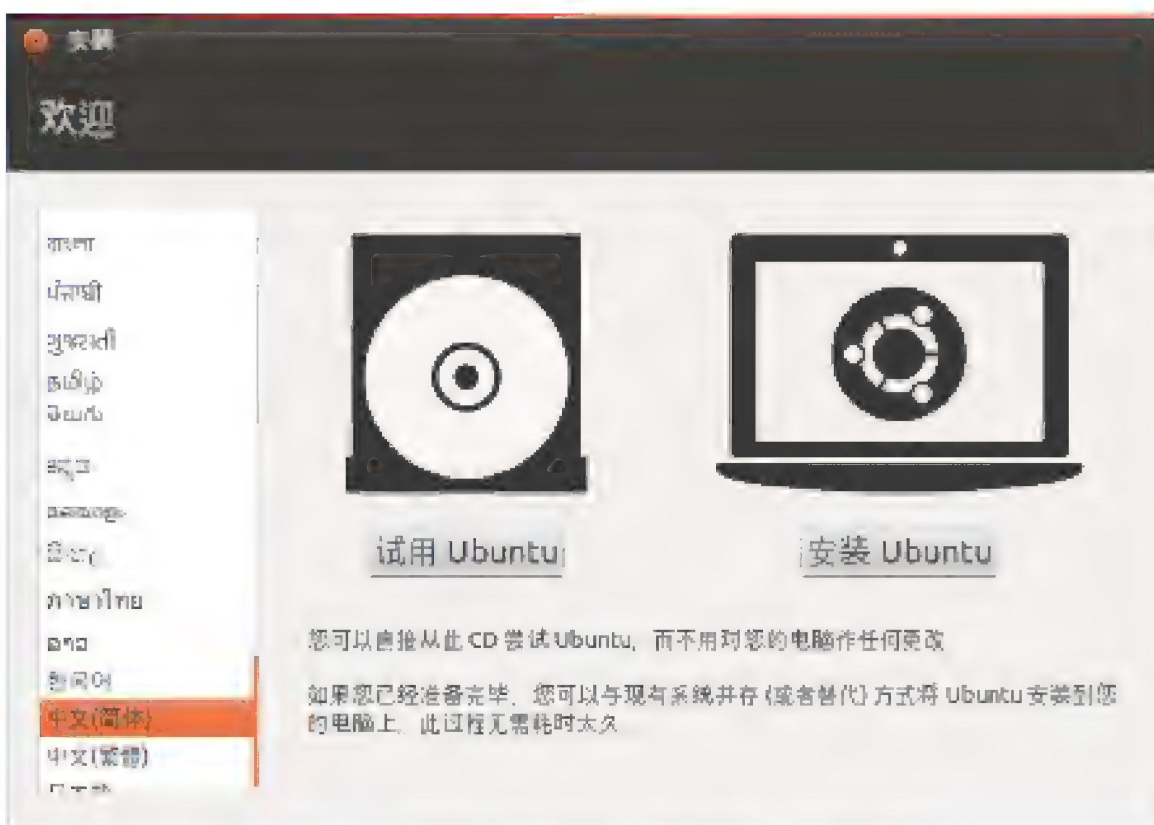


图 1.3　设置好语言的选项界面

单击“安装 Ubuntu”选项之后即进入准备安装界面，如图 1.4 所示，其中提供了对当前计算机的状态检查，包括硬盘空间大小(Ubuntu 12.04 的安装需要最小 4.4GB 的磁盘空间，通常来说最好为其保留 8GB 以上的磁盘空间)、电源选项的检查和联网的检查。

如果此时计算机处于联网状态，则可以勾选“安装中下载更新”选项，Ubuntu 会在安装过程中自动将系统更新到最新状态，如果此时没有连接到网络则该选项不能被选择，但是用户依然可以在联网后通过 Ubuntu 自带的更新管理器或者命令行来完成更新。

此外由于 Ubuntu 使用了一些第三方的软件来完成对应的工作，所以此时应该勾选上“安装这个第三方软件”的选项。

单击“继续”按钮后会出现如图 1.5 所示的磁盘空间选项对话框，在其中可以选择如何对当前计算机中的磁盘空间进行管理和设置，尤其在计算机上有多个操作系统(例如还有 Windows)

的时候该选项对话框是非常重要的；如果只希望在计算机上选择安装一个操作系统，则可以选择上方的“清除整个磁盘并安装 Ubuntu”选项；否则需要选择下方的“其他选项”。



图 1.4　准备安装 Ubuntu 界面



图 1.5　磁盘空间管理和设置

选择“其他选项”，单击“继续”按钮后会看到如图 1.6 所示的磁盘空间管理对话框，在其中会列出计算机中当前对应的硬盘空间。从对话框中可以看到当前计算机上只有一个名称为 /dev/sda 的硬盘，其空间大小为 8.6GB。

下一步应该对这个磁盘空间进行空间划分，单击“新建分区表”按钮会弹出一个如图 1.7 所示的提示对话框，主要用于警告这个操作会导致磁盘上已有的分区表被删除，同时也会导致该硬盘上的数据丢失。

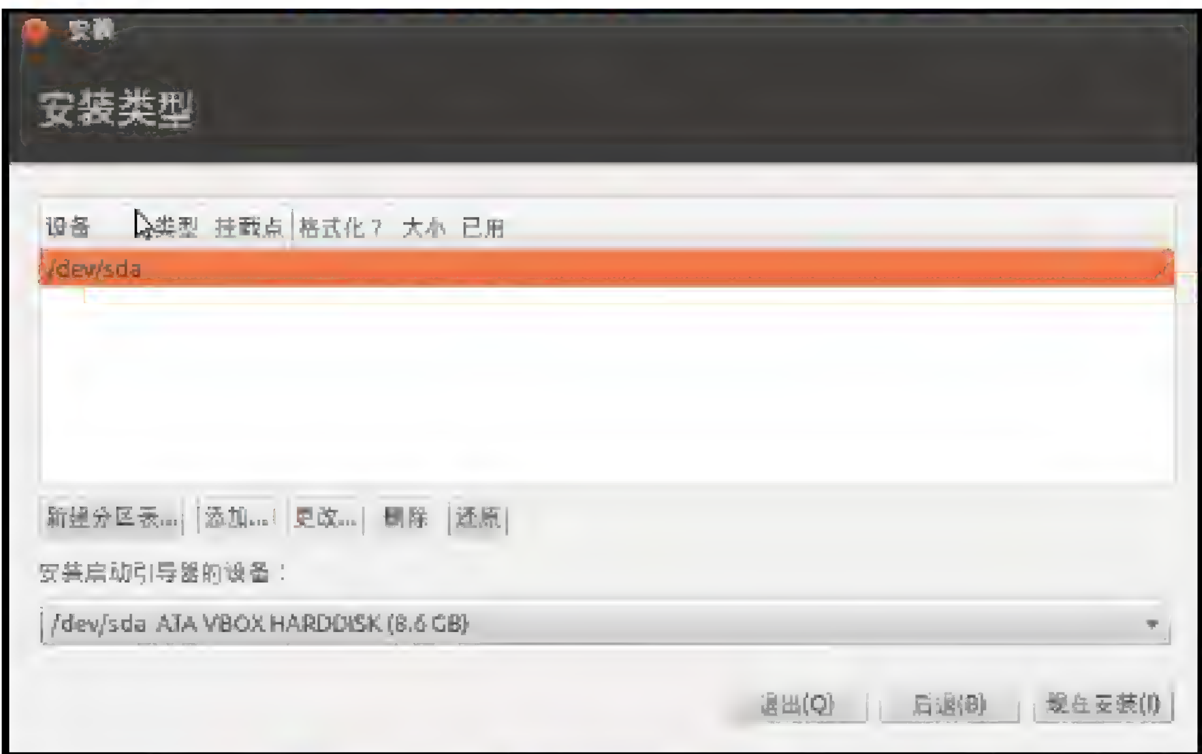


图 1.6　磁盘空间管理对话框



图 1.7　新建分区表警告对话框

单击“继续”按钮后会看到当前硬盘上的分区表已经被抹掉并且成为一个空闲的分区，如图 1.8 所示。

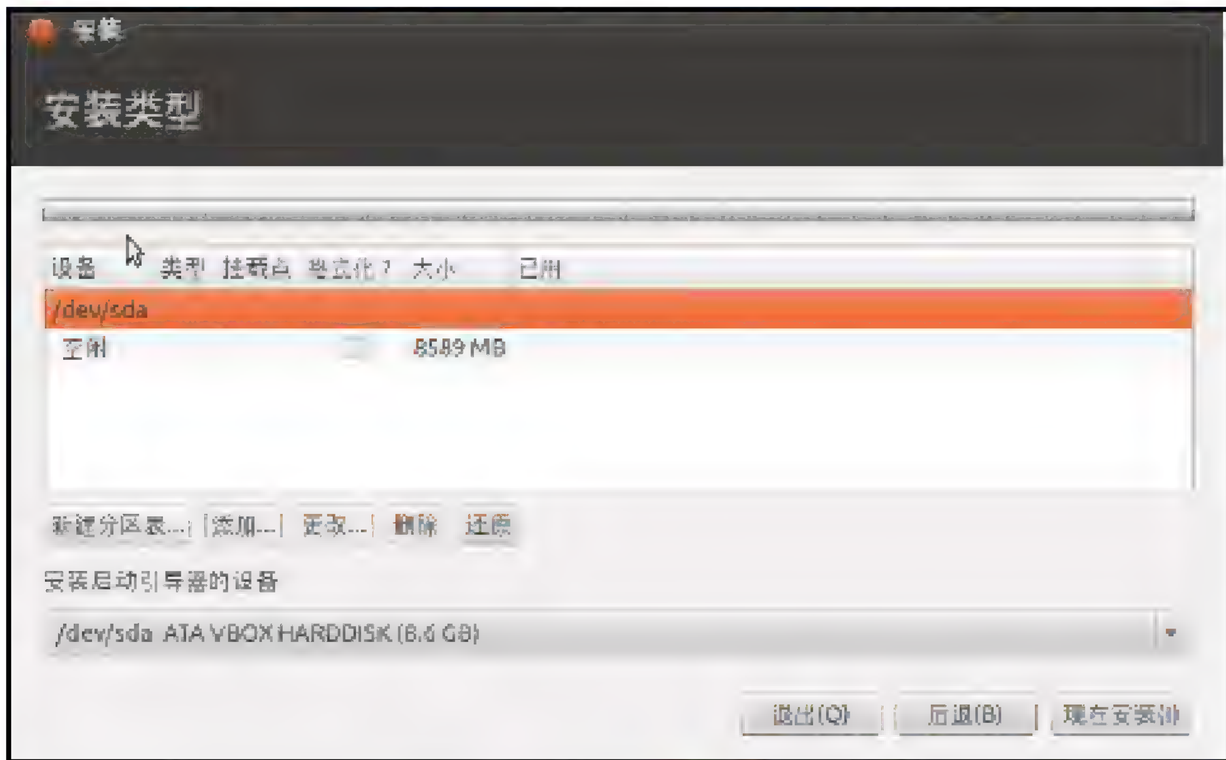


图 1.8　空闲的分区

在如图 1.8 所示的对话框中单击“添加”按钮创建一个新分区，设置该分区为主分区，容量为全部的 8589MB，位于起始位置，用于 Ext4 日志文件系统，单击挂载点下拉菜单选择该挂载点为“/”，在 Linux 系统中，分区并不像 Windows 系统一样排列成 C 盘、D 盘等，而是将操作系统挂载到同一个挂载点，最高一级挂载点为根目录“/”。Linux 允许将一个分区挂载到任何一个文件夹下。设置完成之后的界面如图 1.9 所示。

此时安装系统会出现如图 1.10 所示的提示“是否选择交换分区”提示框，交换分区是(swap)Linux 系统专门用于内存交换的硬盘分区，相当于 Windows 系统中的虚拟内存，交换分区一般设置为内存大小的两倍，在内存不够时，Linux 会将暂不使用的内存数据存放在这个区域。但是在硬件内存比较大的计算机上不需要设置交换空间，直接选择继续即可。

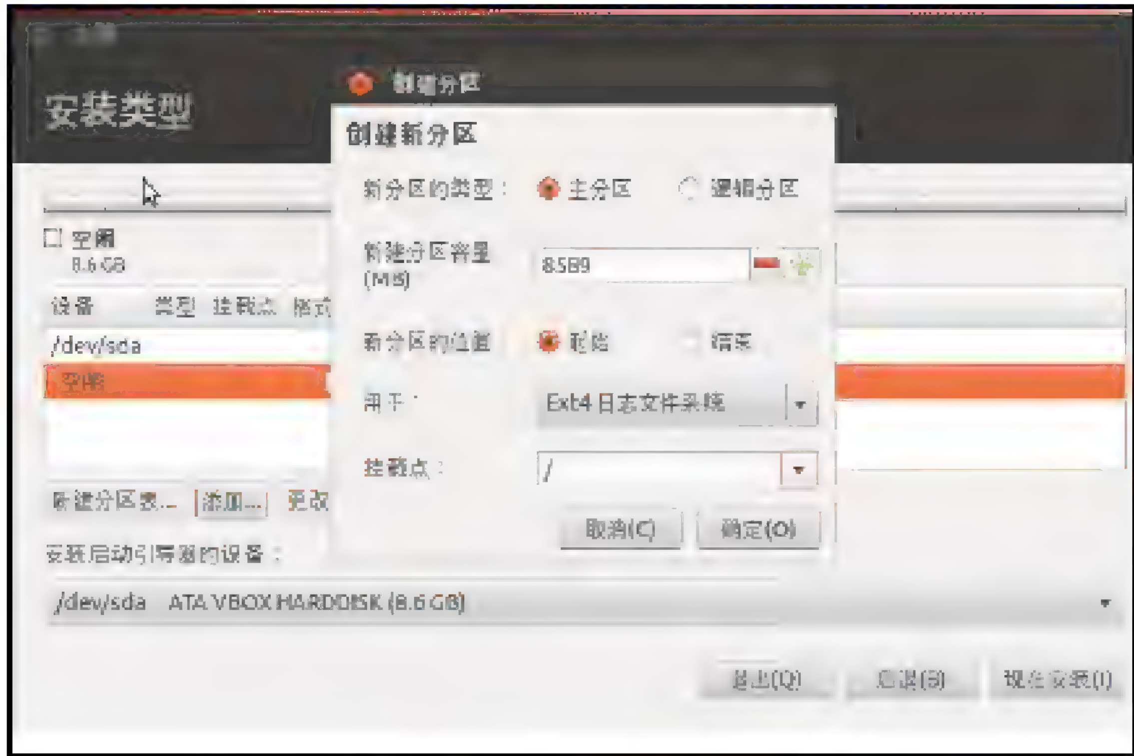


图 1.9　创建新分区

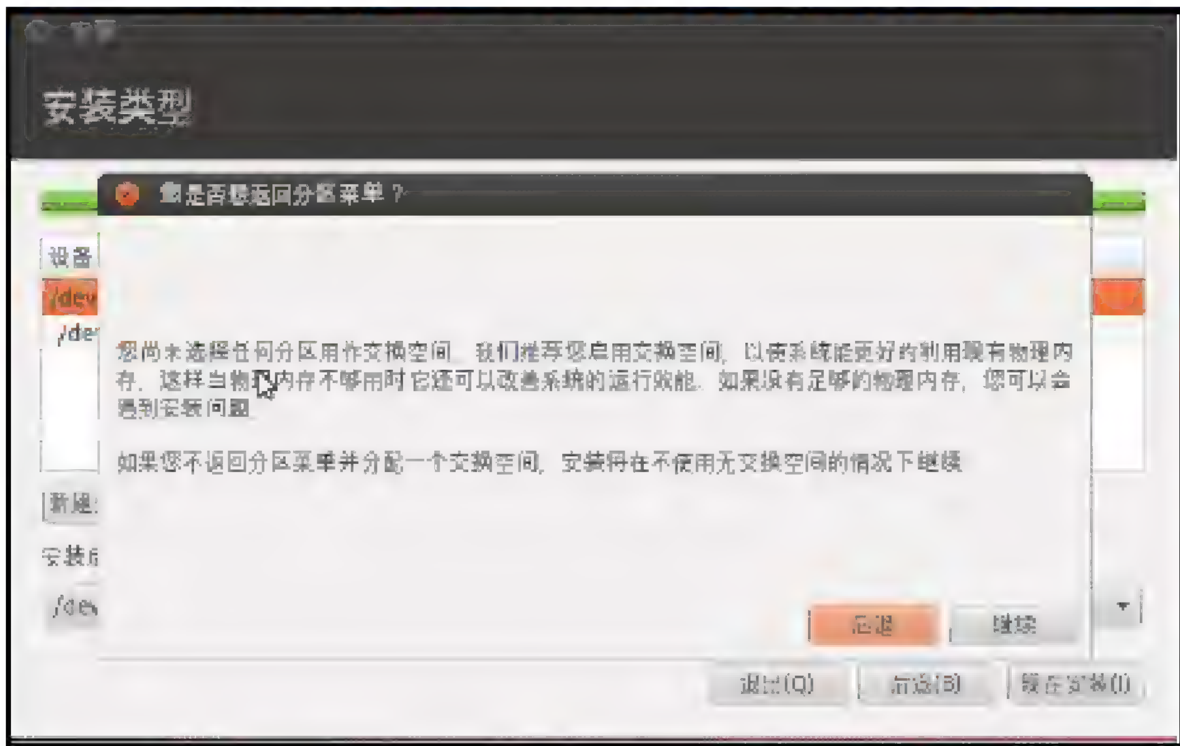


图 1.10　选择交换空间的提示

此时系统会开始建立文件系统并且复制对应的文件，同时在如图 1.11 所示对话框的上方可以进行区域设置，通常来说可以选择上海(shanghai)或者重庆(chongqing)作为中国的区域，这个区域设置会影响语言设置和时间设置。

设置好区域之后单击“继续”按钮，会出现键盘布局设置对话框，其用于设置输入键盘的布局，直接选择汉语即可，如图 1.12 所示。



图 1.11　区域设置

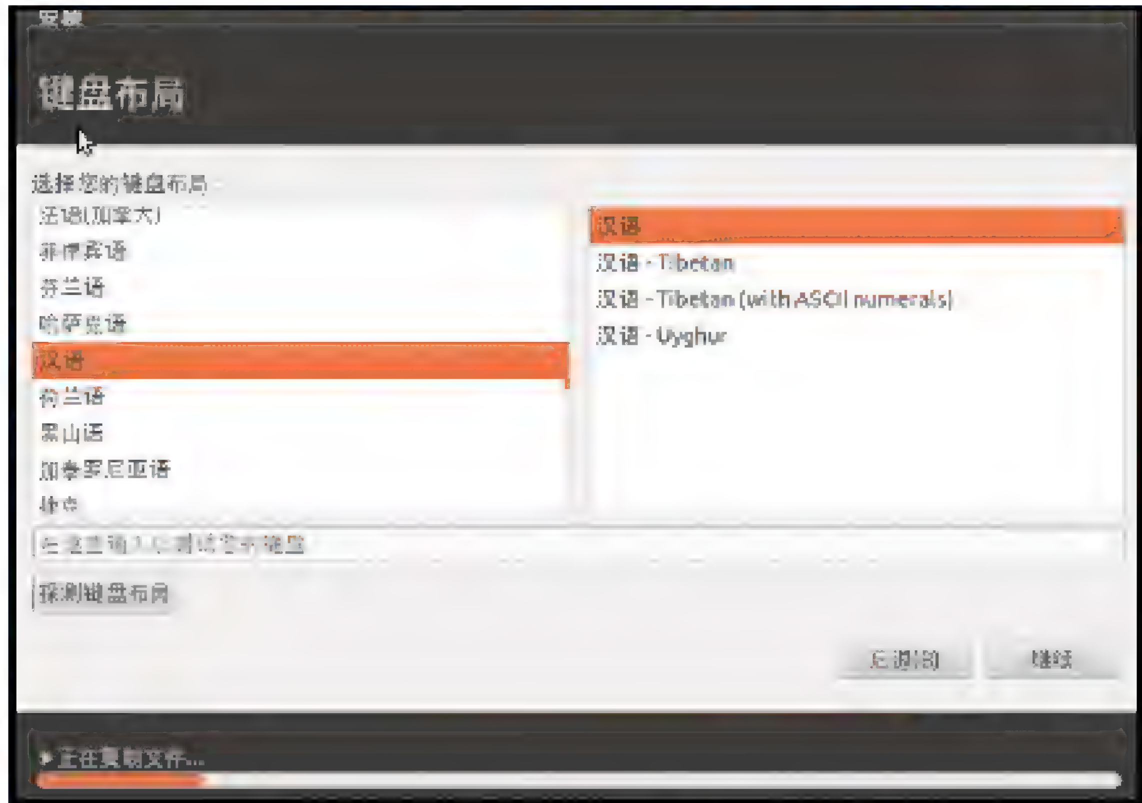


图 1.12　键盘布局设置对话框

接下来的事情即是安心等待直到出现如图 1.13 所示的完成提示框，单击“现在重启”按钮直接重启计算机即可。

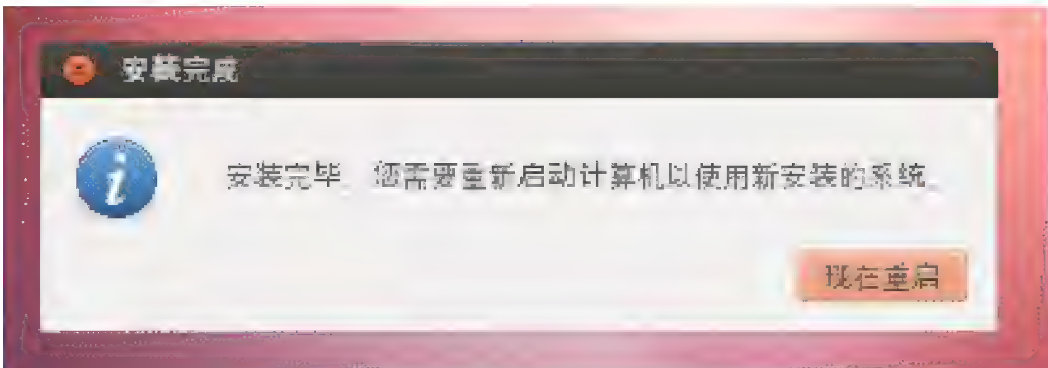


图 1.13　安装完成提示框

再次登录即可看到如图 1.14 所示的 Ubuntu 12.04 的桌面环境。



图 1.14　Ubuntu 12.04 的桌面环境

1.4.3　使用虚拟机

对于 Linux 的初学者，或者说是想把 Linux 操作系统拿来学习，而不是用做服务器的朋友，一种更方便简洁的方式是在虚拟机下安装 Linux。

虚拟机是 Windows 下通过软件模拟的，具有完整硬件系统功能的，运行在一个完全隔离环境中的完整计算机系统的软件。通过虚拟机软件，用户可以在一台物理计算机上模拟出一台或多台虚拟的计算机，这些虚拟机就像真正的计算机那样进行工作，例如你可以安装操作系统、安装应用程序、访问网络资源等。对于用户而言，它只是运行在物理计算机上的一个应用程序，但是对于在虚拟机中运行的应用程序而言，它就像是在真正的计算机中进行工作。目前流行的虚拟机软件主要有 VMware(VMware Workstation)和 Virtual PC，它们都能在 Windows 系统上虚拟出多个计算机，用于安装 Linux、OS/2、FreeBSD 等其他操作系统。如图 1.15 所示为虚拟机软件 VMware Workstation 的初始界面。

单击“New Virtual Machine”，进入新的虚拟计算机安装向导，如图 1.16 所示。单击“下一步”按钮。

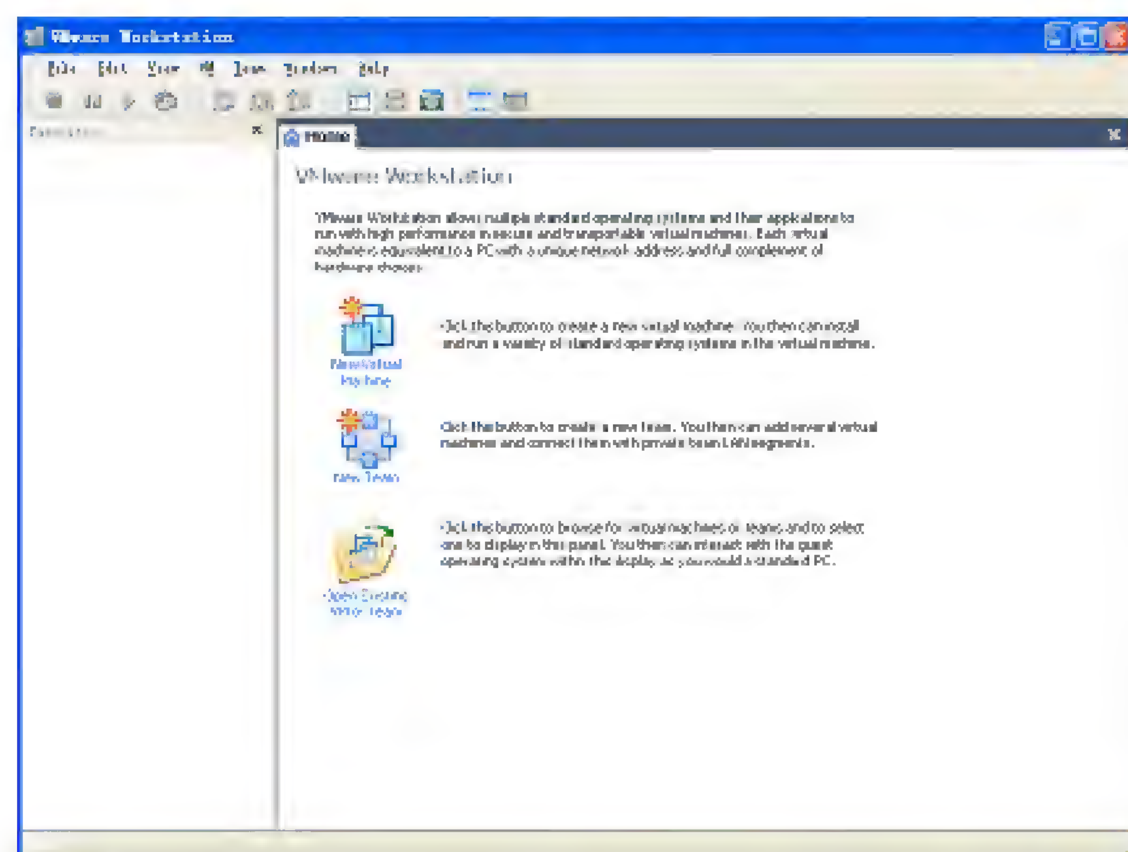


图 1.15　VMWare Workstation 的初始界面



图 1.16　新建虚拟机向导

如图 1.17 所示为选择虚拟机的配置，虚拟机可以选择“Typical”(典型)配置和“Custom”

(自定义)配置。典型配置是指创建一个有普通设备和配置选项的虚拟机，一般均选择此项。选择“Typical”单选按钮，然后单击“下一步”按钮。

接下来在“Select a Guest Operating System”界面中，可以选择将要在虚拟机中安装的计算机操作系统的类型，在这里我们先在“Guest operating system”域中选择“Linux”单选按钮，然后会看到在下方的“Version”下拉列表中有不同的 Linux 版本，这里选择“Red Hat Linux”选项，如图 1.18 所示。然后单击“下一步”按钮。

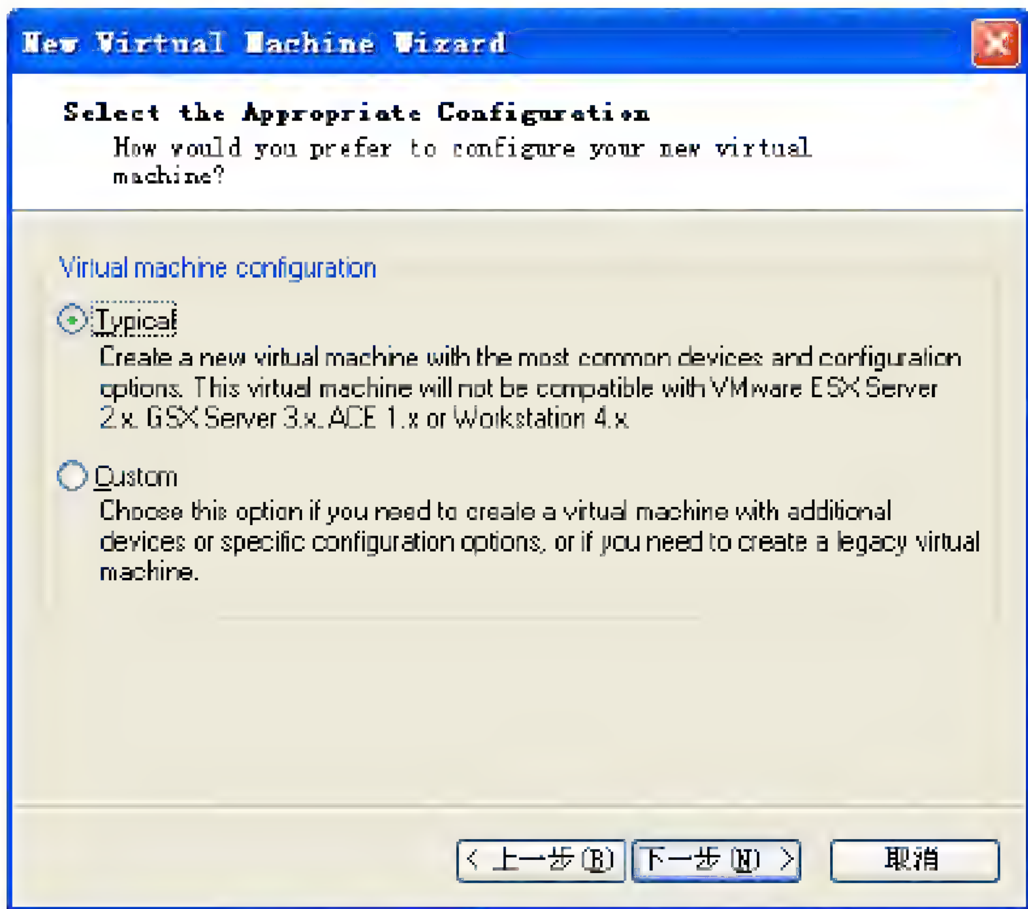


图 1.17　选择虚拟机配置

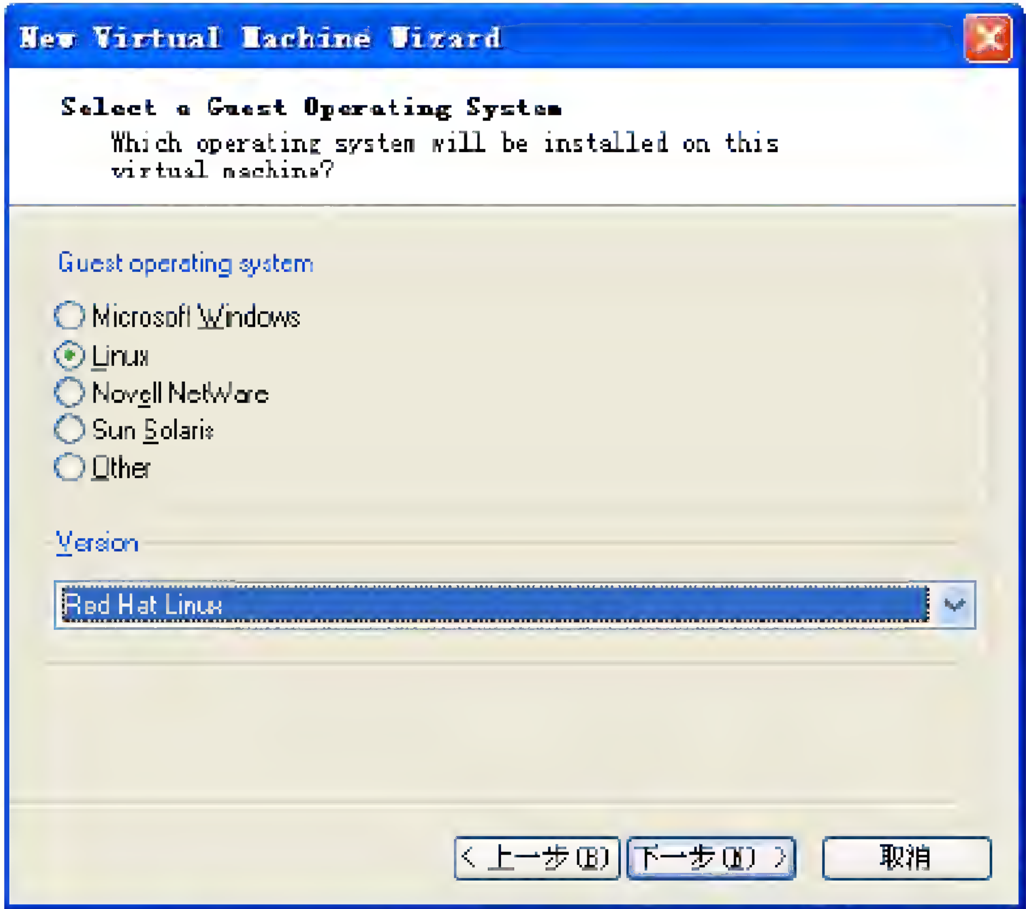


图 1.18　选择虚拟机的操作系统

接下来是设置虚拟机的名称和存储位置，如图 1.19 所示。在“Virtual machine name”文本框中填写虚拟机的名称，用户可随意输入一个合法的名称，这里填写的是“Red Hat Linux”，然后单击“Location”文本框后的“Browse”按钮，选择一个合适的存放虚拟机文件的位置。设置完毕后单击“下一步”按钮。

接下来的步骤是设置虚拟机的网络类型，如图 1.20 所示。在虚拟机网络连接类型的选择对话框中，共有桥接(bridged)、网络地址转换(NAT)、仅主机(host-only)3 种方式。这里选择“Use bridged networking”单选按钮，即使用桥接方式(在安装完成以后是可以更改的)，然后单击“下一步”按钮。

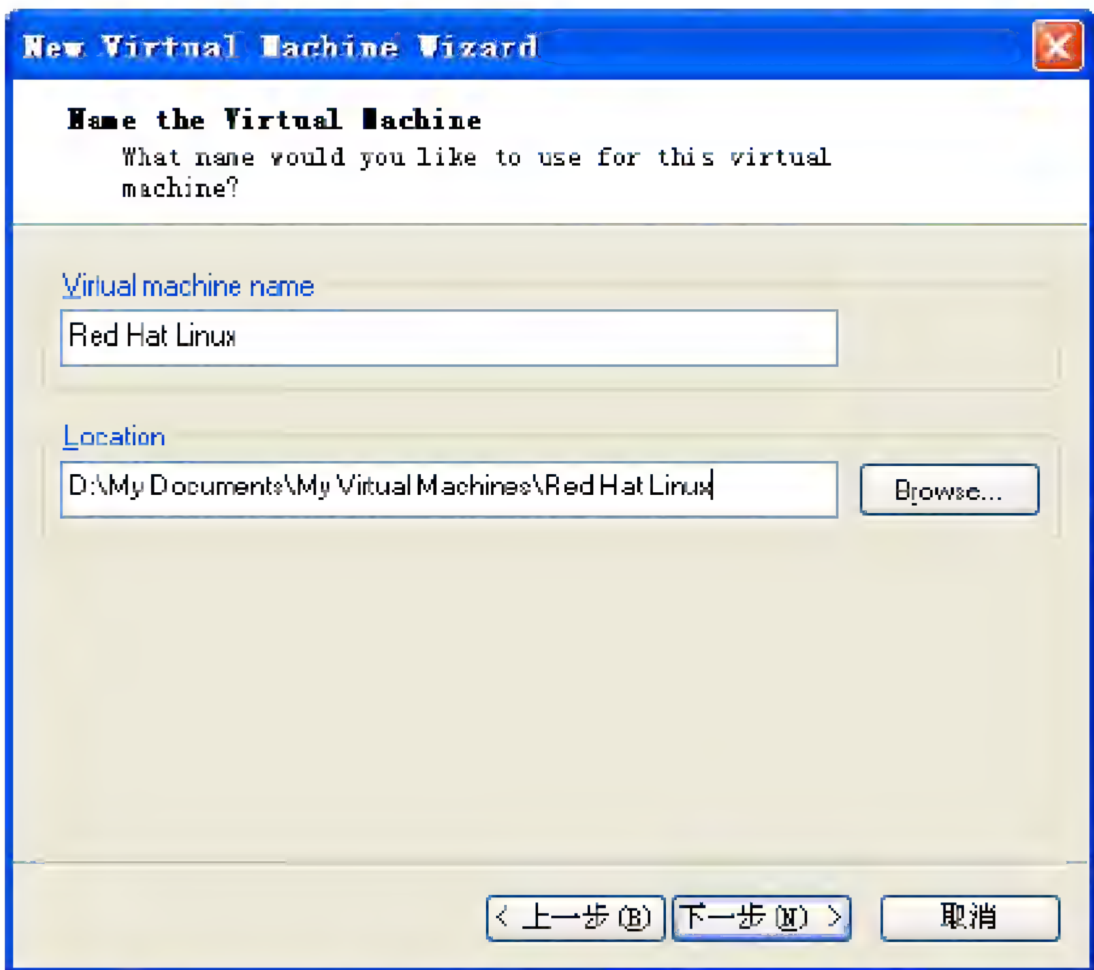


图 1.19　设置虚拟机的名称和存储位置

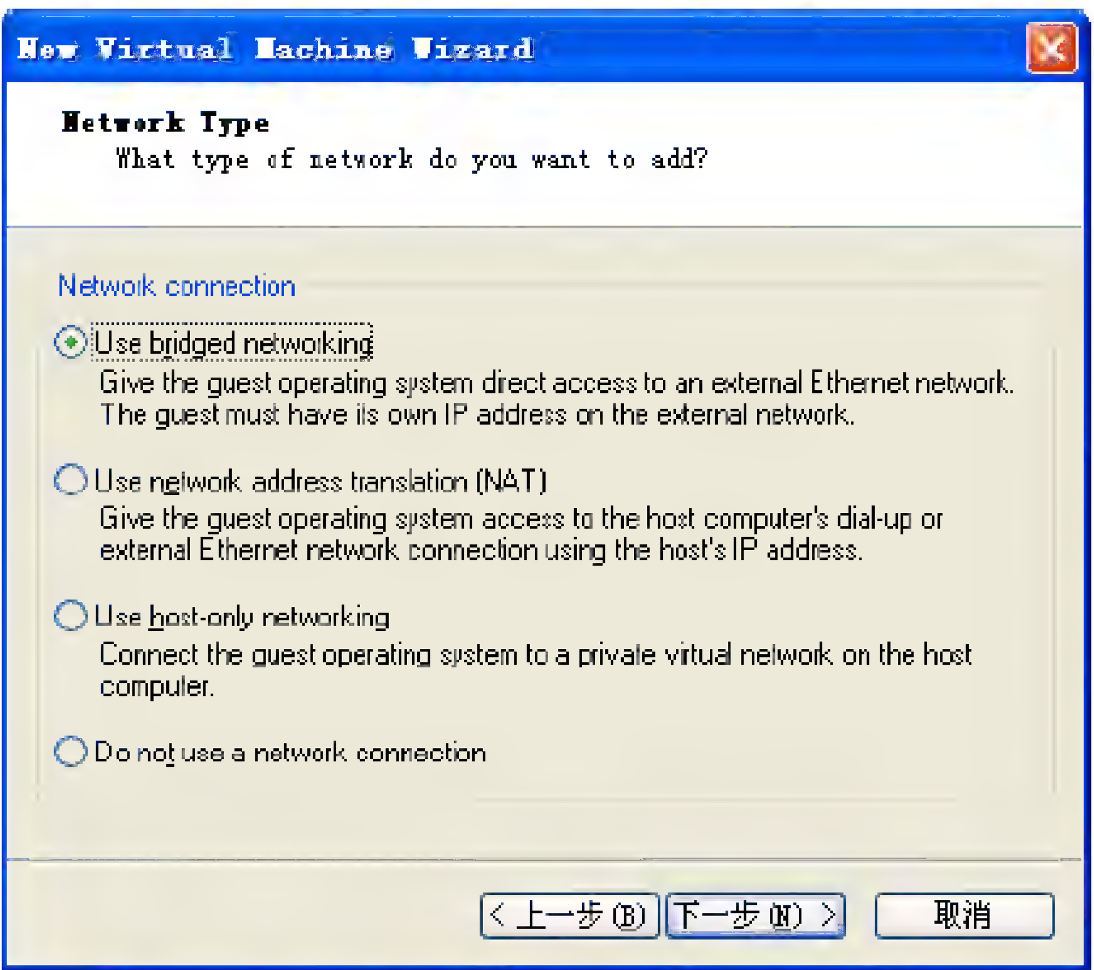


图 1.20　选择虚拟机网络连接的类型

说 明

虚拟机是 Windows 操作系统下的软件，在这个软件的下面再安装操作系统，为了使虚拟机中的操作系统能够使用物理网卡(或者叫网络适配器)而能够同外网(局域网或者是 Internet)通信，就必然涉及虚拟机的网络设置，如上所述，有 bridged、NAT、host-only 3 种方式。

- bridged：这种方式最简单，直接将虚拟网卡桥接到物理网卡上面，和 Linux 下的网卡绑定两个不同 IP 地址类型，实际上是将网卡设置为混杂模式，从而达到侦听多个 IP 的能力。在此种模式下，虚拟机内部的网卡(例如 Linux 下的 eth0)直接连到了物理网卡所在的网络上，使虚拟机和主机处于对等的地位，在网络上是平等的关系。使用这种方式的前提是主机可以得到一个以上的 IP 地址。
- NAT：在这种方式下，主机内部出现了一个虚拟的网卡 VMnet8(默认)，这里的 VMnet8 就相当于连接到内网的网卡，而虚拟机本身则相当于运行在内网上的机器，虚拟机内的网卡(eth0)则独立于 VMnet8。此时，虚拟机软件自带的 DHCP 服务协议会默认加载到 VMnet8 的界面上。更为重要的是，由于虚拟机自带了 NAT 服务，从而提供了从 VMnet8 到外网的地址转换，所以这种情况是一个实实在在的 NAT 服务器在运行，只不过是供虚拟机使用的。很显然，这种方式很适合只有一个外网地址的情况。
- host-only：在这种方式下，没有网络地址转换(NAT)服务，因此虚拟机只能到主机访问，这也是 host-only 的名字的含义。在默认情况下，也会有一个 DHCP 服务加载到 VMnet1 上，这样连接到 VMnet1 上的虚拟机仍然可以设置成 DHCP，方便系统的配置。这种方式最为灵活，可以进行各种网络实验。

接下来是设置磁盘的容量，如图 1.21 所示。在“Disk size”微调框中可以设置虚拟机的最大磁盘容量，安装程序默认为 8GB，表示为该虚拟机文件设置 8GB 的硬盘容量。当然也可以设置得小一些，这里设置为 4GB。最后单击“完成”按钮。

说 明

如图 1.21 所示的步骤中只是对虚拟机的最大磁盘容量进行设置，并没有马上分配这些硬盘空间，只有当虚拟机中存放数据时，系统才会为它分配实际大小的硬盘空间。

此外，还可以设置虚拟机的光驱、网络、声卡等，由于比较简单，在此不再一一赘述。最后，单击“Commands”下的“Start this virtual machine”，即可启动这个虚拟机。

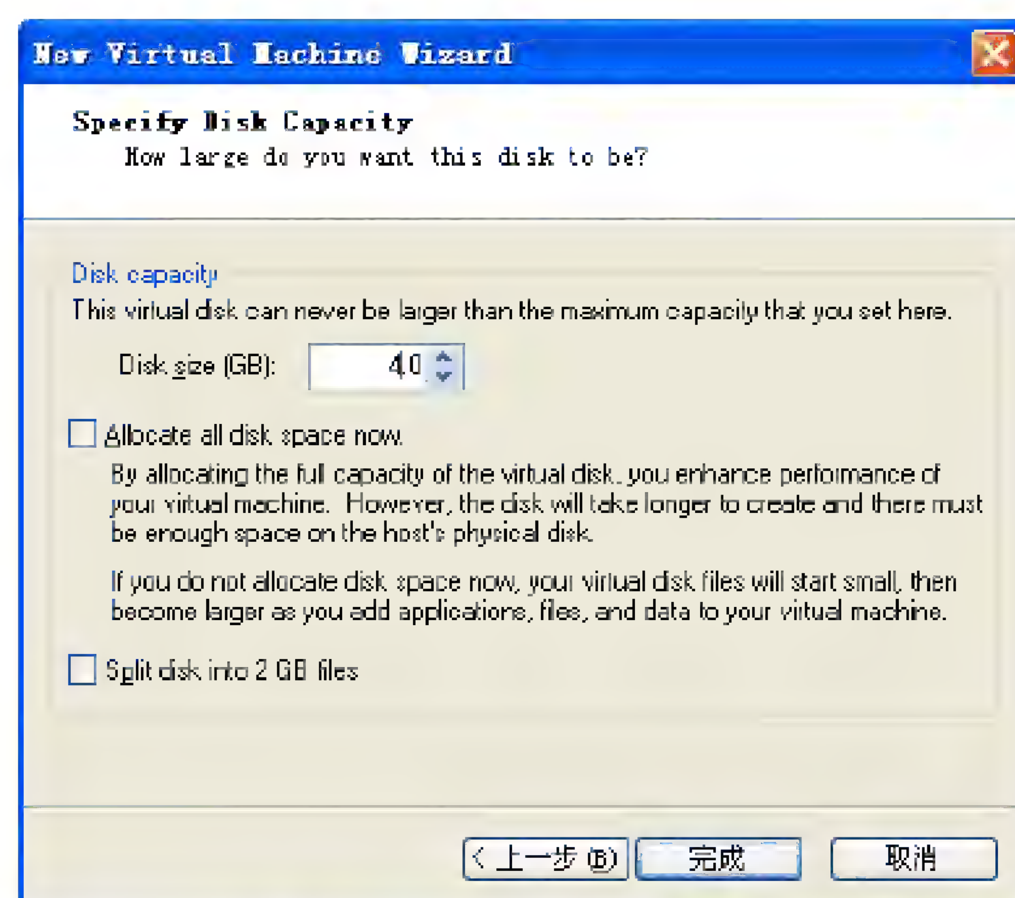


图 1.21　设置磁盘容量

1.5 Shell 的使用

Shell 是 UNIX/Linux 系统的重要组成部分。在 UNIX/Linux 下，Shell 扮演了一个双重角色。虽然它表面上和 Windows 的命令提示符相似，但是它却具备完成执行复杂程序的强大功能。用户不仅可以通过它执行命令、调用 Linux 工具，还可以把 Shell 作为一种编程语音，编写自己的程序。本节向读者介绍 Linux 下 Shell 的简单使用。

1.5.1 Shell 简介

用户登录 Linux 系统时，可以进入基于 X Window 的图形界面系统，如 KDE 或 GNOME。当然，很多工作可以在图形界面环境下完成，但是在服务器应用环境的很多情况下，需要远程连接到服务器进行管理配置，而使用命令行模式进行管理更加方便和简单，因此简单介绍 Linux Shell 的使用是本书必不可少的一部分，但本书毕竟不是 Shell 的专业书籍，在此只能简单介绍 Linux Shell 的基础知识，关于更深入的学习，读者可以参考其他相关书籍。

如果系统设置为不自动启动图形接口，那么用户登录以后得到的就是一个等待输入命令的 Shell 提示符，标识了可以开始发出命令；如果系统设置为自动启动图形系统，那么用户可以选择“开始”→“系统工具”→“终端”命令(不同的 Linux 版本会稍有不同，本书以 Ubuntu 12.04LTS 进行讲解)，运行终端仿真程序，便可以通过在命令提示符后面输入 Linux 命令及参数，进行相应的操作。Shell 终端界面如图 1.22 所示。

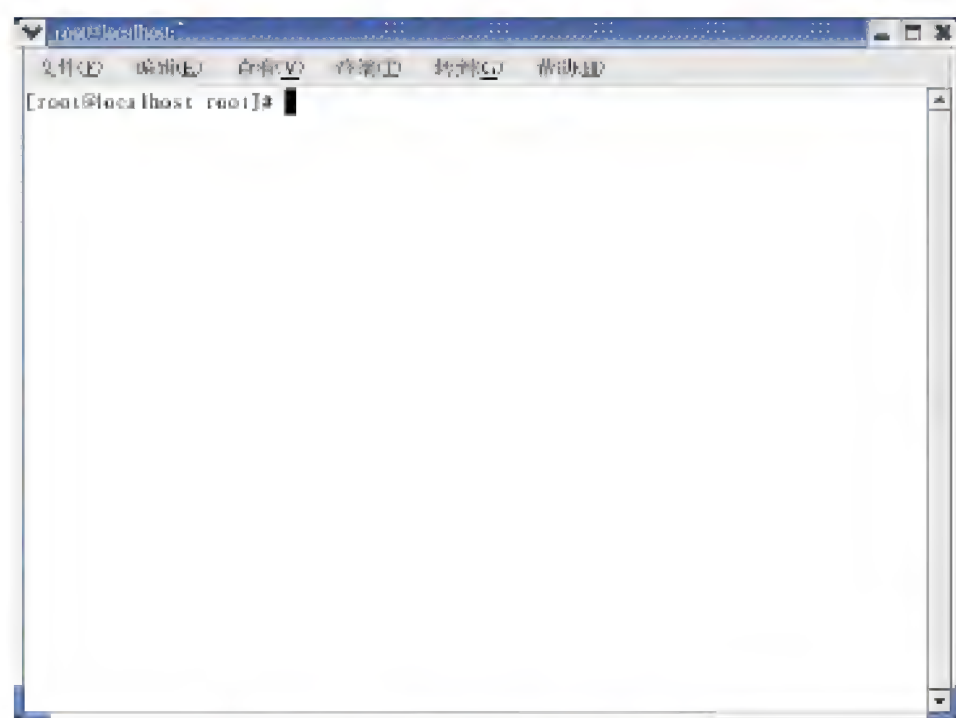


图 1.22　Shell 终端界面

用户登录或运行终端类比程序时，实际就进入了 Shell。那么，Shell 是什么呢？确切地说，Shell 就是一个命令行解释器，它的作用就是遵循一定的语法将输入的命令加以解释并传给操作系统内核，它为用户提供了一个向 Linux 发送请求以便运行程序的接口，用户可以用 Shell 来启动、挂起、停止甚至是编写一些程序。

Shell 本身是一个用 C 语言编写的程序，它是用户使用 Linux 的桥梁。Shell 既是一种命令语言，又是一种程序设计语言。作为命令语言，它互动式地解释和执行用户输入的命令；作为程序设计语言，它定义了各种变量和参数，并提供了许多在高级语言中才具有的控制结构，包括循环和分支。它虽然不是 Linux 系统内核的一部分，但它调用了系统内核的大部分功能来执行程序、创建文档并以并行的方式协调各个程序的运行。因此，对于用户来说，Shell 是最重要的实用程序，深入了解和熟练掌握 Shell 的特性及其使用方法，是用好 Linux 系统的关键。可以说，Shell 使用的熟练程度反映了用户对 Linux 使用的熟练程度。

在 Shell 中，用户使用 Linux 是通过命令来完成所需工作的。一个命令就是用户和 Shell 之间对话的一个基本单位，它是由多个字符组成并以换行结束的字串。Shell 解释用户输入的命令，并翻译成操作系统内核可以执行的指令，系统根据这些指令执行相应的动作。

1.5.2 常见 Shell 的种类

Linux Shell 的种类很多，目前流行的 Shell 包括 ash、bash、ksh、csh、zsh 等，用户可以通过查看/etc/shells 文件中的内容来查看自己主机中当前有哪些种类的 Shell，命令如下(下面是在笔者 Linux 主机中查看信息的结果)：

```
# cat /etc/shells
/bin/sh
/bin/bash
/sbin/nologin
/bin/bash2
/bin/ash
/bin/bsh
/bin/tcsh
/bin/csh
```

使用下面的命令来查看 Linux 当前正在使用的 Shell 类型：

```
# echo $SHELL
```

$SHELL 是一个环境变量，它记录了 Linux 当前用户所使用的 Shell 类型。用户可以通过直接输入各种 Shell 的二进制文件名(因为这些二进制文件本身是可以被执行的)，来进入到该 Shell 下，比如进入 csh 可以直接输入：

```
# /bin/csh
```

这个命令为用户又启动了一个 Shell，这个 Shell 在最初登录的那个 Shell 之后，称为下级的 Shell 或子 Shell。使用命令：

```
# exit
```

可以退出这个子 Shell。使用不同的 Shell 的原因在于它们各自都有自己不同的特点，下面简单介绍 Linux 下各种不同的 Shell 类型的特点。

1. ash

ash Shell 是由 Kenneth Almquist 编写的，是 Linux 中占用系统资源最少的一个小 Shell，它只包含 24 个内部命令，因而使用起来很不方便。

2. bash

bash 是 Linux 系统默认使用的 Shell，它由 Brian Fox 和 Chet Ramey 共同完成，是 Bourne Again Shell 的缩写，内部命令一共有 40 个。Linux 使用它作为默认的 Shell 是因为它具有以下特点：

- 可以使用类似 DOS 下面的 doskey 的功能，用上下方向键查阅和快速输入并修改命令。
- 自动通过查找匹配的方式，给出以某字串开头的命令。
- 包含了自身的帮助功能，在提示符下面键入 help 就可以得到相关的帮助信息。

3. ksh

ksh 是 Korn Shell 的缩写，由 Eric Gisin 编写，共有 42 条内部命令。该 Shell 最大的优点是几乎和商业发行版的 ksh 完全相容，这样就可以在不用花钱购买商业版本的情况下尝试商业版本的性能了。

4. csh

csh 是 Linux 比较大的内核，它由以 William Joy 为代表的共计 47 位作者编成，共有 52 个内部命令。该 Shell 其实是指向/bin/tcsh 这样的一个 Shell，也就是说，csh 其实就是 tcsh。

5. zch

zch 是 Linux 最大的 Shell 之一，由 Paul Falstad 完成，共有 84 个内部命令。如果只是一般的用途，是没有必要安装这样的 Shell 的。

1.5.3　Shell 的简单使用

在 Linux 命令行中输入的第一个字必须是一个命令的名字，第二个字是命令的选项或参数，命令行中的每个字必须由空格或 Tab 隔开，格式如下：

```
$ 命令 选项 参数
```

或者：

```
# 命令 选项 参数
```

说　明

提示符“$”和“#”区分了用户的不同权限，“$”表示普通用户权限，而“#”代表的是根用户(超级用户)权限。

选项是包括一个或多个字母的代码，它前面有一个减号(减号是必要的，Linux 用它来区别选项和参数)，选项可用于改变命令执行的动作的类型。

命令行实际上是一个可以编辑的文本缓冲区，在按 Enter 键之前，可以对输入的文本进行编辑。比如利用“BackSpace”键可以删除刚输入的字符，可以进行整行删除，还可以插入字符，使得用户在输入命令(尤其是复杂命令)时，若出现输入错误，无须重新输入整个命令，只要利用编辑操作，即可改正错误。

利用上箭头可以重新显示刚执行的命令，利用这一功能可以重复执行以前执行过的命令，而无须重新输入该命令。

bash 保存着以前输入过的命令的列表，该列表被称为命令历史表。按上箭头，便可以在命令行上逐次显示各条命令。同样，按下箭头可以在命令列表中向下移动，这样可以将以前的各条命令显示在命令行上，用户可以修改并执行这些命令。

在一个命令行中还可以置入多个命令，用分号将各个命令隔开，这些命令将按顺序执行。也可以在几个命令行中输入一个命令，用反斜杠将一个命令行持续到下一行。总之，Shell 命令

行的输入灵活方便，当用户熟悉 Linux 的命令之后，会更加体会到这一点。

1.5.4 通配符

在 Shell 中除使用普通字符外，还可以使用一些具有特殊含义和功能的字符，称为通配符，在使用它们时应注意其特殊的含义和作用范围。

Shell 的通配符主要用于模式匹配，如文件名匹配、路径名搜索、字串查找等。常用的通配符有“*”、“?”和括在方括号“[ ]”中的字符序列等，用户可以在作为命令参数的文件名中包含这些通配符，构成一个所谓的“模式串”，以在执行过程中进行模式匹配。这 3 个通配符的含义分别如下：

- “*”代表任意长度的字串，例如“L*”匹配以 L 开头的任意字串。但应注意，文件名中的圆点(.)和路径名中的斜线(/)必须是显式的，即不能用通配符替代它们。例如“*”不能匹配.c，而“.*”才可以匹配.c。
- “?”代表任何单个字符。
- “[ ]”指定了模式串匹配的字符范围，只要文件名中“[ ]”处的字符在指定的范围之内，那么这个文件名就与该模式串匹配。方括号中的字符范围可以由字符串组成，也可以由表示限定范围的起始字符、终止字符及中间连字符“-”组成。例如，f [a-d]与 f [abcd]的作用相同。

Shell 将把与命令行中指定的模式串相匹配的所有文件名都作为命令的参数，形成最终的命令，然后再执行这个命令。如果目录中没有与指定的模式串相匹配的文件名，那么 Shell 将使用此模式串本身作为参数传给命令(这正是命令中出现特殊字符的原因所在)。

表 1.2 列举了这些通配符的具体实例及含义。

表 1.2 通配符的含义

模式串举例	含　义
*	当前目录下所有文件的名称
Text	当前目录下所有文件名中含有 Text 字串的文件的名称
[ab-dm]*	当前目录下所有以 a,b,c,d,m 开头的文件的名称
[ab-dm]?	当前目录下所有以 a,b,c,d,m 开头且后面只跟有一个字符的文件的名称
/usr/bin/??	目录/usr/bin/下所有名称长度为两个字符的文件的名称

需要注意的是，中间连字符“-”仅在方括号内有效，表示字符范围，若在方括号外面，就成为普通字符了。而“*”和“?”则只在方括号外有效，若出现在方括号之内，它们也失去通配符的能力，成为普通字符了。例如，模式 L[*?]abc 中只有一对方括号是通配符，而“*”和“?”均为普通字符，因此，它匹配的字串只能是 L*abc 和 L?abc。

注 意

由于“*”、“?”和“[ ]”对于 Shell 来说具有比较特殊的意义，因此在文件名中不应出现这些字符，特别是在目录名中不要出现它们，否则可能导致 Shell 无穷递归地进行匹配。

1.5.5 引号

在 Shell 中引号分为 3 种：单引号、双引号和反引号。

1. 单引号

由单引号引起来的字符都作为普通字符出现。特殊字符用单引号引起来以后，也会失去原有意义，而只作为普通字符解释。例如下面的一系列命令：

```
# string='$PATH'            #定义字符变量 string，其值为字符串$PATH
# echo $string              #显示字符变量 string 的值
$PATH
```

可见，单引号中的“$”保持了其本身的含义，作为普通字符出现。而在一般情形下，“$”符号的含义是引用变量的值，PATH 本身是一个 Linux 下的环境变量，其值是一系列的目录，当用户运行某个程序时，Linux 在这些目录下进行搜寻。可以使用下面的命令查看变量 PATH 的值：

```
#echo $PATH
```

读者可仔细体会它与上面例子中的不同。

对于本书中的注释，若无特殊说明，均采用以下约定：

说 明

在命令行、Shell 程序、Makefile(将在第 5 章介绍)中，我们使用“#”注释符，“#”后面为注释内容，读者注意要区别于 Linux 命令提示符“#”；而在 C 程序代码中，我们使用“/* */”的方式作为注释符。

2. 双引号

双引号的作用与单引号类似，区别在于它没有那么严格。单引号告诉 Shell 忽略所有的特殊字符，而双引号只要求忽略大多数。具体来说，引在双引号中的 3 种特殊字符不被忽略：“$”，“\”和“`”，即双引号会解释字符串的特别意义，而单引号则直接使用字符串。如果使用双引号将字符串赋给变量并反馈它，实际上与直接反馈变量并无差别。如果要查询包含空格的字符串，经常会用到双引号。

我们看下面的例子：

```
# x=*                    #定义字符变量 x，其值为字符*
# echo $x                #显示变量 x 的值
Anaconda-ks.cfg Desktop install.log install.log.sys.log
# echo '$x'              #使用单引号
$x
# echo "$x"              #使用双引号
*
```

从这个例子中，可以清楚地看出无引号、单引号和双引号之间的区别。

第一种情况，显示变量 x 的值。由于 x 的值，即字符*匹配了当前目录(root 目录)下的所有文件名，故显示变量 x 的值时，即显示了当前目录的所有文件名。

第二种情况，使用了单引号。单引号中的字符保持其本身的含义，这种情况最简单。

最后一种情况，使用了双引号。双引号告诉 Shell 在引号内照样进行变量名替换，所以 Shell 把$x 替换为*，因为双引号中不做文件名替换(忽略掉了非特殊字符)，所以就把*作为要显示的值传递给 echo 命令，作为 echo 命令的参数。

另外，从例子中还可以看到 Shell 赋值的先后次序：Shell 先做变量替换，然后做文件名替换，最后把这些替换值作为参数传递给命令。

作为复习，读者可以自行演练并思考本章最后习题中的第 8 题。

3. 反引号

反引号(`)字符所对应的键一般位于键盘的左上角，不要将其同单引号(')混淆。反引号引起来的字串被 Shell 解释为命令行，在执行时，Shell 首先执行该命令行，并以它的标准输出结果取代整个反引号(包括两个反引号)部分。例如：

```
# pwd
/home/zhangfan
# string="current directory is `pwd`"       #定义字符变量 string
# echo $string                               #显示变量 string 的值
current directory is /home/zhangfan
```

Shell 执行 echo 命令时，首先执行`pwd`中的命令 pwd，并将输出结果/home/xyz 取代`pwd`部分，最后输出替换后的整个结果。

利用反引号的这种功能可以进行命令置换，即把反引号引起来的执行结果赋值给指定变量。再例如：

```
# today=`date`
# echo Today is $today
Today is Wed Apr 25 15:13:28 CST 2009
```

另外，反引号还可以嵌套使用。但需要注意的是，嵌套使用时内层的反引号必须用反斜线(\)将其转义。例如：

```
# num=`echo The number of users is \`who | wc -l\``        #反引号嵌套使用
# echo $num                                                 #显示变量 num 的值
The number of users is 3
```

Shell 首先执行内层嵌套的“who | wc -l”命令，再将其结果“3”作为参数传递给下一个命令 echo(外层反引号中的 echo)。

1.5.6 注释符

在 Shell 编程或 Linux 的配置文档中，经常要对某些正文行进行注释，以增加程序的可读性。在 Shell 中以字符#开头的正文行表示注释行。

1.6 Linux 常用命令

在 Linux 操作系统中，各式各样的命令有成百上千个，有的命令用户会经常用到，有的则可能很少甚至几乎不会用到。本节将向读者简单介绍 Linux 下的常用命令，而 Linux 本身庞大的命令集，则需要用户在不断的应用和实践中日积月累了。

1.6.1 与目录相关的命令

当用户在 Linux 的 Shell 终端中执行相关的操作时，目录的操作是最为常见的。与目录操作相关的常用命令包括 pwd、cd、mkdir、rmdir 等。

1. pwd 命令

格式：pwd；功能：显示当前目录的绝对路径。

2. cd 命令

格式：cd [目录路径名]；功能：切换到指定目录。

cd 命令是在 Linux 命令行中使用最为频繁的命令之一。例如下面的命令：

```
#cd   /usr/bin        #从当前工作目录转到/usr/bin 子目录
#cd ..                #从当前工作目录转到上一层子目录
```

3. mkdir 命令

格式：mkdir [目录路径名]；功能：创建一个新的子目录，子目录的路径名作为参数。

4. rmdir 命令

格式：rmdir [-p] 目录路径名；功能：删除空目录。若有参数 p，当子目录被删除后，若当前目录也成为空目录的话，则一起删除。

1.6.2 与文件相关的命令

文件是 Linux 系统中非常重要的一个概念，可以说，用户在 Linux 下任何操作都可以看作是对某种文件的操作。下面简要介绍与文件相关的常用命令。

1. ls 命令

格式：ls [选项] [文件|目录]；功能：显示指定目录中的文件和子目录信息。当不指定目录时，显示当前目录下的文件和子目录信息。

主要选项：-l，查看当前目录下文件或子目录的详细信息。这些信息包括文件或目录的权限、链接数、所有者、用户组、文件大小、日期和文件名。关于这些，将在本书的第 6 章进行详细讲解。

2. cat 命令

格式：cat [选项] 文件列表；功能：显示文本文件的内容。

主要选项：-n (number)，表示在每一行前显示行号。

3. more 命令

格式：more filename；功能：分屏显示文本文件 filename 的内容。在文件的内容较多时，屏幕不能一次完全显示，这个命令就起到作用了。

4. cp 命令

格式：cp [选项] 源文件 目标文件；功能：将一个文件复制到另一文件，或将数个文件复制到另一目录。

主要选项：-r，若源文件中含有目录名，则将目录下的文件也都依序复制到目的地。

5. mv 命令

格式：mv [选项] 源文件 目标文件；功能：将一个文件移至另一个文件，或将数个文件移至另一个目录。

主要选项：-i，若目的地已有同名文件，则先询问是否覆盖旧文件。

6. rm 命令

格式：rm [选项] [文件|目录]；功能：删除文件及目录。

主要选项：-i，删除前逐一询问确认；-r，将目录及以下的文件也逐一删除；-f，即使原文件属性设为只读，也直接删除，无须逐一确认。

7. chmod 命令

格式：chmod [选项] [文件|目录]；功能：改变文件的权限属性(详见第 6 章)。

8. tar 命令

格式：tar [选项] [tar 的文件名] [文件列表]；功能：压缩、解压缩 tar 格式的压缩包，制作备份、恢复备份文件等。

常用的选项：-c，建立一个新的 tar 文件；-v，显示运行过程信息；-z，使用 gzip；-t，查看压缩文件的内容；-f，文件名称；-x，解压缩 tar 文件；-M，制作存放于多个备份介质上的备份档案。

9. 获得帮助

(1) man 命令

格式：man 命令名；功能：显示指定命令的帮助信息。例如：

```
#man ls         #获得 ls 命令的帮助
```

(2) --help 选项

格式：--help 命令名；功能：显示指定命令的帮助信息。例如：

```
#cat --help          #显示 cat 命令的帮助信息
#ls --help | more    #分页显示 ls 命令的帮助信息
```

1.6.3 与网络服务相关的命令

在配置与使用 Linux 的网络服务时，需要用到与网络服务相关的命令。这些常用命令包括

ping、ifconfig、netstat、telnet 等。

1. ping 命令

格式：ping 网络主机地址；功能：同 Windows 下的 ping 命令，查看本机到网络上某一主机是否能够通信。

2. ifconfig 命令

格式：ifconfig 网卡名称；功能：配置网卡信息，如 IP 地址、子网掩码和默认网关。

3. netstat 命令

格式：netstat [参数列表]；功能：显示本机网络状态。

常用参数：-a 或-all，表示显示所有连接中的套接字 Socket(将在第 11 章介绍)。

4. telnet 命令

格式：telnet 网络主机地址；功能：同 Windows 下的 telnet 命令，远程登录到网络上的某一服务器主机。

1.7 本章小结

本章向读者介绍了 Linux 操作系统的基础知识，Linux 系统的特点，详细讲解了 Linux 的各种安装方式，Linux Shell 的基本使用，最后列出了笔者在多年使用 Linux 操作系统的过程中最常用的命令，请读者务必先熟练掌握这些命令。相信初学者阅读完本章会对 Linux 有一个比较清晰的认识，这会为日后的学习打下扎实的基础。

实战演练

1. 试在虚拟机软件 VMware Workstation 下使用 Ubuntu 12.04LTS 的 ISO 镜像文件安装 Linux 操作系统。
2. 在 Shell 中使用命令查看本机用户名和 Linux 的内核版本。
3. 使用命令查看 Linux 当前正在使用的 Shell 类型，并从当前的 shell 切换至 csh 下。
4. 使用命令在/home 目录下创建一个新的空目录 temp。
5. 使用命令将/root 目录下的 hello 文件复制到/home/temp 目录下。
6. 使用命令改变/home/temp 目录下的 hello 文件的权限属性，将该文件的权限属性设置为文件所有者可读可写可执行、用户组可读可执行、其他组用户可读可执行。
7. 试将虚拟机的网络类型设置为 host-only 方式，并在 Shell 终端下使用 ping 命令查看 Linux 能否与外网通信。
8. 在 Linux 下若存在环境变量$HOME=Hello World，当运行命令 echo '$HOME'时的输出是什么？运行命令 echo "$HOME"时的输出又是什么？并在 Shell 中检验你的猜测。
9. 试使用命令行的方式将 Linux 主机 IP 设为 192.168.1.100，子网掩码为 255.255.255.0，默认网关设为 192.168.1.1。
10. 试使用 Linux 命令查看本机的网络状态信息。

第 2 章

C语言编程基础

C 语言是国际上广泛使用的计算机高级编程语言，C 语言最初用于描述和支持 UNIX 系统，后来逐渐被广大程序员所接受，成为备受欢迎的编程语言。在其后的发展过程中，C 语言不断吸收计算机方面新的成果，使该语言逐渐完善起来。作为 Linux 系统的开发语言，C 语言在 Linux 编程开发中扮演着重要的角色。本章将向读者详细讲解 C 语言的相关编程基础知识。



本章内容：

◎ C 语言产生的历史背景。
◎ C 语言的特点。
◎ C 语言的基本数据类型。
◎ 运算符与表达式。
◎ C 程序的 3 种基本结构。
◎ C 语言中的数据输入与输出。
◎ 函数、数组、指针、结构体和共用体、链表。
◎ 位运算符和位运算。
◎ C 语言的预处理命令。

2.1 C 语言的历史背景

C 语言的原型是 A 语言(ALGOL 60 语言)。1963 年，剑桥大学将 A 语言发展成为 CPL(Combined Programming Language)语言。1967 年，剑桥大学的 Matin Richards 对 CPL 语言进行了简化，于是产生了 BCPL 语言。1969 年，美国贝尔实验室的 Ken Thompson 将 BCPL 进行了修改，提炼出它的精华，并为它起名为“B 语言”。并且他用 B 语言写了第一个 UNIX 操作系统。而在 1973 年，美国贝尔实验室的 D.M.Ritchie 在 B 语言的基础上最终设计出了一种新的语言，他取了 BCPL 的第二个字母作为这种语言的名字，这就是 C 语言。

为了使 UNIX 操作系统得以推广，1977 年 D.M.Ritchie 发表了不依赖于具体机器系统的 C 语言编译文本《可移植的 C 语言编译程序》。即著名的 ANSI C。1978 年由 AT&T(美国电话电报公司)贝尔实验室正式发表了 C 语言。同时由 B.W.Kernighan 和 D.M.Ritchit 合著了著名的《THE C PROGRAMMING LANGUAGE》一书。通常简称为《K&R》，也有人称之为《K&R》标准。但是，在《K&R》中并没有定义一个完整的标准 C 语言，后来由美国国家标准协会(American National Standards Institute)在此基础上制定了一个 C 语言标准，于 1983 年发表。通常称之为 ANSI C。

1987 年，随着微型计算机的日益普及，出现了许多 C 语言版本。由于没有统一的标准，使得这些 C 语言之间出现了一些不一致的地方。为了改变这种情况，美国国家标准研究所(ANSI)为 C 语言制定了一套 ANSI 标准，成为现行的 C 语言标准。

1990 年，国际化标准组织 ISO(International Standard Organization)接受了 87 ANSI C 为 ISO C 的标准(ISO 9899-1990)。1994 年，ISO 修订了 C 语言的标准。目前流行的 C 语言编译系统大多是以 ANSI C 为基础进行开发的，但不同版本的 C 编译系统所实现的语言功能和语法规则略有差别。

2.2 C 语言的特点

C 语言之所以能被世界计算机界广泛接受，正是由于它自身具备的突出特点，从语言体系和结构上讲，它与 Pascal、ALGOL 60 等语言相类似，是结构化程序设计语言。归纳起来，C 语言具有下列特点：

- C 语言是中级语言，它把高级语言的基本结构和语句与低级语言的实用性结合起来。例如，位、字节和地址是计算机最基本的工作单元，而 C 语言可以像汇编语言一样对这三者进行操作。
- C 语言是结构式语言。结构式语言的显著特点是代码及数据的分隔化，即程序的各个部分除了必要的信息交流外彼此独立，这种结构化方式可使程序层次清晰，便于使用、维护及调试。C 语言是以函数作为程序的模块单位，用户可方便地调用这些模块，并可通过多种循环、条件语句控制程序流向，从而使程序完全结构化。

➢ C 语言功能齐全。C 语言提供多种的数据类型，能用来实现各种复杂的数据结构，例如通过引入了指针来使程序效率更高。另外 C 语言包含了 34 种运算符，丰富的运算符使其具有强大的计算功能和逻辑判断功能。
➢ C 语言适用范围广。C 语言适用于多种操作系统(如 DOS、UNIX)，也适用于多种机型。在对操作系统、系统应用程序及需要对硬件进行操作时，都选择使用 C 语言。而且，用 C 语言编写的程序，只要稍加修改就可移植到不同型号的计算机上。

C 语言对程序员要求也高。程序员用 C 语言编写程序会感到限制少、灵活性大、功能强，但较其他高级语言在学习上要困难一些。上面只介绍了 C 语言的一般特点。相信通过后续章节的实践，读者能够体会到 C 语言更多其他特点。

2.3 C 语言的基本数据类型

在计算机中，数据的性质和表示方式可能不同。所以需要将相同性质的数据归类，并用一定数据类型描述。任何数据对用户都呈现常量和变量两种形式。常量是指程序在运行时其值不能改变的量。常量不占内存，在程序运行时它作为操作对象直接出现在运算器的各种寄存器中。变量是指在程序运行时其值可以改变的量。变量的功能就是存储数据。C 语言的基本数据类型包括整型、实型和字符型。下面我们将分别进行介绍。

2.3.1 整型

整型即整数数据类型。分整型常量和整型变量两部分进行介绍。

1. 整型常量

整型常量就是整常数。在 C 语言中，使用的整常数有八进制、十六进制和十进制 3 种。

(1) 十进制整型常数是数码为 0～9 的十进制数字串，没有前缀。例如，237、-568、65 535、1 627。

(2) 八进制整常数必须以 0 开头，即以 0 作为八进制数的前缀。数码取值为 0～7。八进制数通常是无符号数。例如，015(十进制为 13)、0101(十进制为 65)、0177777(十进制为 65535);

(3) 十六进制整常数的前缀为 0X 或 0x。其数码取值为 0～9，A～F 或 a～f。例如，0X2A(十进制为 42)、0XA0(十进制为 160)、0XFFFF(十进制为 65535)。

2. 整型变量

在 C 语言中，整型变量有 6 种类型：基本整型、无符号基本整型、短整型、无符号短整型、长整型和无符号长整型。

表 2.1 列出了 ANSI 标准定义的各种整型变量所分配的内存字节数，以及数的表示范围。

表 2.1　ANSI 标准规定的整型变量

类　型	类型说明符	字节	数 值 范 围
基本整型	[signed] int	2	−32 768～32 767，即-2^{15}～$(2^{15}-1)$

(续表)

类　型	类型说明符	字节	数 值 范 围
无符号基本整型	unsigned [int]	2	0～65 535，即 0～$(2^{16}-1)$
短整型	[signed] short [int]	2	−32 768～32 767，即-2^{15}～$(2^{15}-1)$
无符号短整型	unsigned short [int]	2	0～65 535，即 0～$(2^{16}-1)$
长整型	[signed] long [int]	4	−214 783 648～214 783 647，即-2^{31}～$(2^{31}-1)$
无符号长整型	unsigned long [int]	4	0～4 294 967 295，即 0～$(2^{32}-1)$

2.3.2　实型

实型即实数数据类型，也称为浮点型。分实型常量和实型变量两部分进行介绍。

1. 实型常量

实型常量也称为实数或者浮点数。在C语言中，实数只采用十进制。它有两种形式：十进制小数形式和指数形式。

(1) 十进制小数形式：由数码 0～9 和小数点组成。如 0.0、25.0、5.789、0.13、5.0、300.、-267.8230 等，均为合法的实数。注意，必须有小数点。

(2) 指数形式：由十进制数，加阶码标志“e”或“E”及阶码(只能为整数，可以带符号)组成。例如：

2.1E5(等于 2.1×10^{5})

3.7E−2(等于 3.7×10^{-2})

0.5E7(等于 0.5×10^{7})

-2.8E−2(等于-2.8×10^{-2})

2. 实型变量

实型变量分为单精度(float 型)、双精度(double 型)和长双精度(long double 型)3 类。实型数据的属性见表 2.2。

表 2.2　实数基本类型表

类型说明符	比特数(字节数)	有 效 数 字	数 的 范 围
float	32(4)	6～7	10^{-37}～10^{38}
double	64(8)	15～16	10^{-307}～10^{308}
long double	128(16)	18～19	10^{-4931}～10^{4932}

实型变量定义的格式和书写规则与整型相同。例如：

```
float x,y;       /*x,y 为单精度实型量*/
double a,b,c;    /*a,b,c 为双精度实型量*/
```

由于实型变量是由有限的存储单元组成的，因此能提供的有效数字总是有限的。实型数据有时存在舍入误差。

2.3.3 字符型

文字处理是计算机的一个重要应用领域，这个应用领域的程序必须能够使用和处理字符形式的数据。字符型数据存储的是字符的 ASCII 码，一个字符占一个字节。

比如字符‘x’的十进制 ASCII 码是 120，字符‘y’的十进制 ASCII 码是 121。对字符变量 a、b 赋予'x'和'y'值：

```
a = 'x';
b='y';
```

实际上是在 a、b 两个单元内存放 120 和 121 的二进制代码：

a：

0	1	1	1	1	0	0	0

b：

0	1	1	1	1	0	0	1

所以也可以把字符看成是整型量。C 语言允许对整型变量赋予字符值，也允许对字符变量赋予整型值。在输出时，允许把字符变量按整型格式输出，也允许把整型量按字符格式输出，如程序 2.1。

【程序 2.1】向字符变量赋予整数：test1.c。

```
#include <stdio.h>
main()
{
    char a,b;                       /*定义两个字符型变量*/
    a=120;                          /*给字符型变量赋整数值*/
    b=121;
    printf("%c,%c\n",a,b);          /*以字符格式输出变量值*/
printf("%d,%d\n",a,b);              /*以整型格式输出变量值*/
}
```

程序运行结果如下：

```
x,y
120,121
```

可以看到，当以字符格式输出变量值时，结果是 ASCII 码值 120 和 121 分别对应的字符 x 和 y，而以整型格式输出变量值时，结果是一个整型数。

说 明

printf()函数称为格式输出函数，它的调用形式为：

```
printf ("格式控制字符串", 输出项表);
```

对于 printf()函数格式串中的格式符，当格式符为“%c”时，对应输出的变量值为字符，当格式符为“%d”时，对应输出的变量值为整数。详细格式请参见 2.6.3 小节。

除了以上形式的字符常量外，还有一种特殊的字符常量叫转义字符。转义字符以反斜线“\”开头，后跟一个或几个字符。转义字符具有特定的含义，不同于字符原有的意义，故称“转义”

字符。转义字符主要用来表示那些用一般字符不便于表示的控制代码。表 2.3 列出了 C 语言中常用的转义字符。

表 2.3 常用的转义字符及其含义

转 义 字 符	转义字符的含义	ASCII 代码
\n	回车换行	10
\t	横向跳到下一制表位置	9
\b	退格	8
\r	回车	13
\f	走纸换页	12
\\	反斜线符“\”	92
\'	单引号符	39
\"	双引号符	34
\a	鸣铃	7
\ddd	1～3 位八进制数所代表的字符	
\xhh	1～2 位十六进制数所代表的字符	

广义地讲，C 语言字符集中的任何一个字符均可用转义字符来表示。表中的\ddd 和\xhh 正是为此而提出的。ddd 和 hh 分别为八进制和十六进制的 ASCII 代码。如\101 表示字母"A" ，\102 表示字母"B"，\134 表示反斜线，\XOA 表示换行等。

字符变量用来存储字符，一个字符型变量占用一个字节的内存容量，字符变量的类型说明符是 char。字符变量类型定义的格式和书写规则都与整型变量相同。例如：

```
char a,b;
```

字符串常量是由一对双引号引起来的字符序列。例如："CHINA" ，"C program"， "$12.5" 等都是合法的字符串常量。

注意，字符串常量和字符常量是不同的量，它们之间主要有以下区别：

- 字符常量由单引号引起来，字符串常量由双引号引起来。
- 字符常量只能是单个字符，字符串常量则可以包含一个或多个字符。
- 可以把一个字符常量赋予一个字符变量，但不能把一个字符串常量赋予一个字符变量(在 C 语言中没有相应的字符串变量)，但是可以用一个字符数组来存放一个字符串常量(数组将在 2.10 节予以介绍)。
- 字符常量占一个字节的内存空间，字符串常量占的内存字节数等于字符串中字节数加 1。增加的一个字节中存放字符"\0"(ASCII 码为 0)，它是字符串结束的标志。例如，字符串 "C program" 在内存中所占的字节为：

C		p	r	o	g	r	a	m	\0

字符常量'a'和字符串常量"a"虽然都只有一个字符，但在内存中的情况是不同的。

'a'在内存中占一个字节，可表示为：



"a"在内存中占二个字节，可表示为：



2.4 运算符与表达式

C 语言中有很多种运算符和表达式，如算术运算、赋值运算、逗号运算、自增、自减、关系运算、逻辑运算、位运算、条件运算等。正是由于C语言具有丰富的多种类型的表达式，才得以体现出C语言所具有的表达能力强、使用灵活、适应性好的特点。本节向读者介绍算术、赋值和逗号运算符，其他的运算符将在本章中结合有关内容陆续进行介绍。

2.4.1 算术运算符与算术表达式

1. 基本的算术运算符

C 语言的基本算术运算符如表 2.4 所示。

表 2.4 算术运算符

运算符	含　义	运算对象个数	结 合 方 向	例　子
+	加法运算或取正值运算	双目、单目运算符	自左至右	a+b,　+5
-	减法运算或取负值运算	双目、单目运算符	自左至右	a-b,　-5
*	乘法运算	双目运算符	自左至右	a*b
/	除法运算	双目运算符	自左至右	a/b
%	模运算(求余运算)	双目运算符	自左至右	5%7

这里需要说明以下几点：

(1) “+”、“-”作为单目运算符(如-x，-5)时，具有左结合性。作为单目运算符使用时其优先级高于双目运算符。

(2) 除法运算符“/”在使用时要注意数据类型。参与运算量均为整型时，结果也为整型，舍去小数。如果运算量中有一个是实型，则结果为双精度实型。例如，20/7，-20/7 的结果均为整型，小数全部舍去；而 20.0/7 和-20.0/7 由于有实数参与运算，其结果也为实型。

(3) 求余运算符(模运算符)“%”要求参与运算的量均为整型，其结果等于两数相除后的余数。

2. 算术表达式

C 语言的算术表达式由常量、变量、函数、运算符和圆括号组成。例如：3+5，3.2*5.6+7，-5*(18%4+9)，x/(y+z)，sin(x)+sin(y)。它们都是合法的算术表达式。使用算术表达式时必须注意两个问题：一是双目运算符两侧的运算对象类型必须一致；二是括号可以改变表达式的运算

顺序，先计算括号中的表达式，再计算括号外的表达式。

2.4.2 赋值运算符与赋值表达式

赋值运算符记为“=”，由“=”连接的式子称为赋值表达式。其一般形式为：

```
变量 = 表达式
```

赋值表达式的功能是先计算表达式的值，再赋予左边的变量。赋值运算符具有右结合性。例如：

```
pi=3.14;          /*将常数 3.14 赋值给变量 pi*/
S= (a+b)*h/2;     /*先计算算术表达式(a+b)*h/2 的值，再赋值给变量 S*/
L=2*pi*r;         /*先计算算术表达式 2*pi*r 的值，再赋值给变量 L*/
```

按照C语言规定，任何表达式在其末尾加上分号就构成语句，如上面的例子中就构成了C语言的赋值语句。

另外，如果赋值运算符两边的数据类型不相同，系统将自动进行类型转换，即把等号右边的类型转换成左边的类型，具体规定如下：

- 实型赋予整型，舍去小数部分。
- 整型赋予实型，数值不变，但以浮点形式存放，即增加小数部分(小数部分的值为 0)。
- 字符型赋予整型，由于字符型为一个字节，而整型为两个字节，故将字符的 ASCII 码值放到整型量的低八位中，高八位为 0。
- 整型赋予字符型，只把低八位赋予字符量。

程序 2.2 说明了 C 语言赋值运算中的类型转换规则，代码如 test2.c 所示。

【程序 2.2】赋值运算中的类型转换规则：test2.c。

```
#include <stdio.h>
main()
{
    int a,b=322;
    float x,y=8.88;
    char c1='k',c2;
    a=y;                    /*给整型变量赋实型值*/
    printf("a=%d\n",a);
    x=b;                    /*给实型变量赋整型值*/
    printf("x=%f\n",x);
    a=c1;                    /*给整型变量赋字符型值*/
    printf("a=%d\n",a);
    c2=b;                    /*给字符型变量赋整型值*/
    printf("c2=%c\n",c2);
}
```

程序运行结果如下：

```
a=8
x=322.000000
a=107
c2=B
```

可以看到，由于 a 为整型，所以赋予实型变量 y 值 8.88 后只取整数部分 8。x 为实型，赋予整型变量 b 值 322 后增加了小数部分。字符型变量 c1 赋予 a 变为整型，整型量 b 赋予 c2 后取其低八位成为字符型(b 的低八位为 01000010，即十进制 66，按 ASCII 码对应于字符 B)。

在赋值符“=”之前加上其他二目运算符可构成复合赋值符。如 +=,-=,*=, /=,%=,<<=,>>=,&=,^=,|=。

它们等价于各自相应的运算符，例如：

- a+=5 等价于 a=a+5。
- x*=y+7 等价于 x=x*(y+7)。
- r%=p 等价于 r=r%p。

复合赋值符这种写法，对初学者可能不习惯，但十分有利于编译处理，能提高编译效率并产生质量较高的目标代码。

2.4.3 逗号运算符与逗号表达式

在 C 语言中逗号“,”也是一种运算符，称为逗号运算符，其功能是把两个表达式连接起来组成一个表达式，称为逗号表达式。其一般形式为：

```
表达式 1, 表达式 2
```

逗号表达式的求值过程是，分别求出两个表达式的值，并以表达式 2 的值作为整个逗号表达式的值，如程序 2.3 中的代码。

【程序 2.3】逗号表达式的运算规则：test3.c。

```
#include <stdio.h>
main()
{
    int a=2,b=4,c=6,x,y;
    y=((x=a+b),(b+c));              /*用逗号表达式对 y 赋值*/
    printf("y=%d, x=%d",y,x);       /*显示 x、y 的值*/
}
```

程序运行结果如下：

```
y=10, x=6
```

从结果可以看出，y 等于整个逗号表达式的值，也就是表达式 2 的值 10，而 x 是第一个表达式的值 6。

2.5 C 程序的 3 种基本结构

算法的实现过程是由一系列操作组成的，这些操作之间的执行次序就是程序的控制结构。计算机科学家证明：任何简单或复杂的算法都可以由顺序、选择和循环这 3 种基本结构组合而成。所以这 3 种结构就被称为程序设计的 3 种基本结构。

2.5.1　顺序结构

顺序结构的程序设计是最简单的，程序中的各个操作按照它们出现的先后顺序执行，其流程如图 2.1 所示。

图中 S1 和 S2 表示两个处理步骤。整个顺序结构只有一个入口点和一个出口点。这种结构的特点是程序从入口点开始，按顺序执行所有操作，直到出口点，所以称为顺序结构。不论程序中包含了什么样的结构，程序的总流程都是顺序结构。程序 2.4 给出一个顺序结构程序设计的例子。

【程序 2.4】设有变量 x 和 y，编程序实现两个变量值的互换。

实现两个变量值互换的方式有很多种，本例中我们使用中间变量 t 来实现这个功能。先把 x 的值保存在变量 t 中，即 t=x；然后执行 x=y；此时，虽然 x 的值被 y 的值取代，但 x 的值事先已经保存在另一个变量 t 中，所以在使用 y=t 时；就可以把原 x 的值赋给 y。从而实现 x、y 值的互换，程序流程如图 2.2 所示。代码实现见 test4.c。

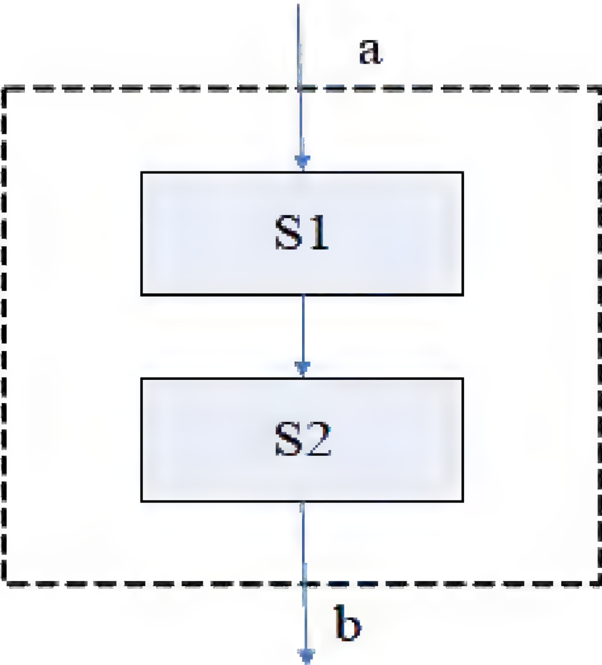


图 2.1　顺序结构

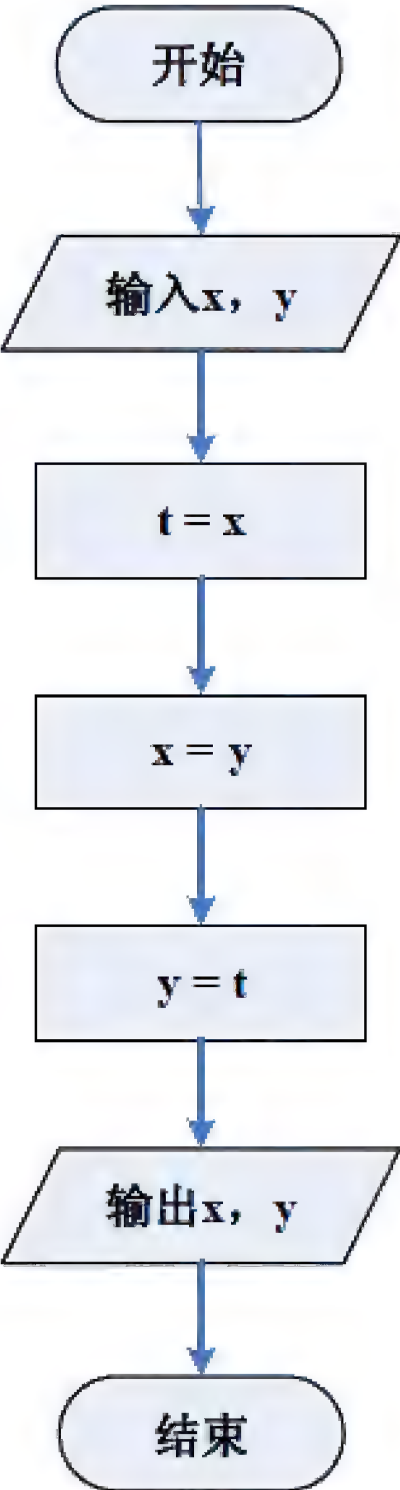


图 2.2　两个变量值互换

test4.c：

```
#include <stdio.h>
main()
{
        int x,y,t;
        printf("Enter x and y:\n");     /*提示用户输入数据*/
        scanf("%d %d",&x,&y);           /*通过格式输入 scanf 读取输入值*/
        t = x;                          /*交换算法*/
        x = y;
        y = t;
```

```
        printf("x=%d, y=%d\n",x,y);   /*显示交换结果*/
}
```

程序运行结果如下(□表示空格，↙表示回车):

```
Enter x and y:
10□5↙
x=5, y=10
```

可以看到，变量 x 和 y 的值进行了互换。

2.5.2 选择结构

选择程序结构用于判断给定的条件，根据判断的结果来控制程序的流程。在选择结构中，程序的处理步骤出现了分支，它需要根据某一特定的条件选择其中的一个分支执行。选择结构有单选择、双选择和多选择 3 种形式。

单选择结构是最简单的选择结构，如图 2.3 所示，如果条件满足则执行 S1，否则向下流到出口处。也就是说，当条件不满足时，什么也没执行。C 语言用 if 语句实现这种功能。其一般形式为:

```
if (表达式) 语句序列
```

if 语句的执行过程与 if else 相似(稍后将会看到)，只是在判断条件为“假”时，直接跳过语句序列，执行 if 的下一条语句。

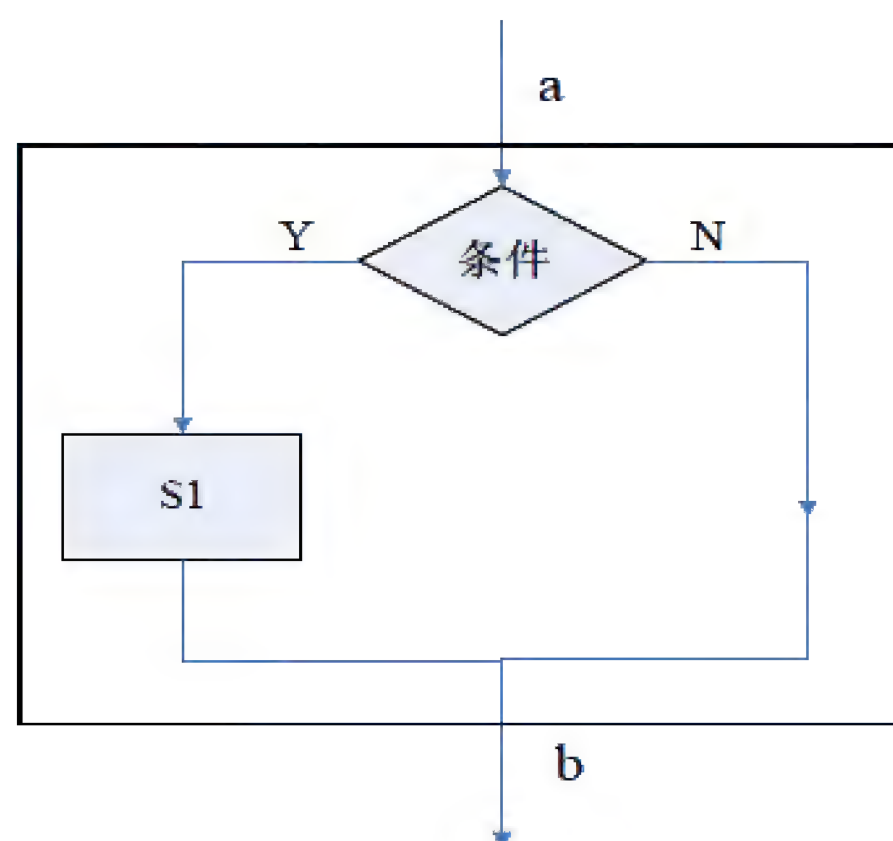


图 2.3 单选择结构

双选择结构如图 2.4 所示，程序流程出现了两个可供选择的分支，如果条件满足就执行 S1 处理，否则执行 S2 处理。两个分支中只能选择一条且必须选择一条执行，但不论选择了哪一条分支执行，最后流程都一定到达结构的出口点处。C 语言用 if else 语句实现这种功能。其一般形式为:

```
if (表达式) 语句 1
else   语句 2
```

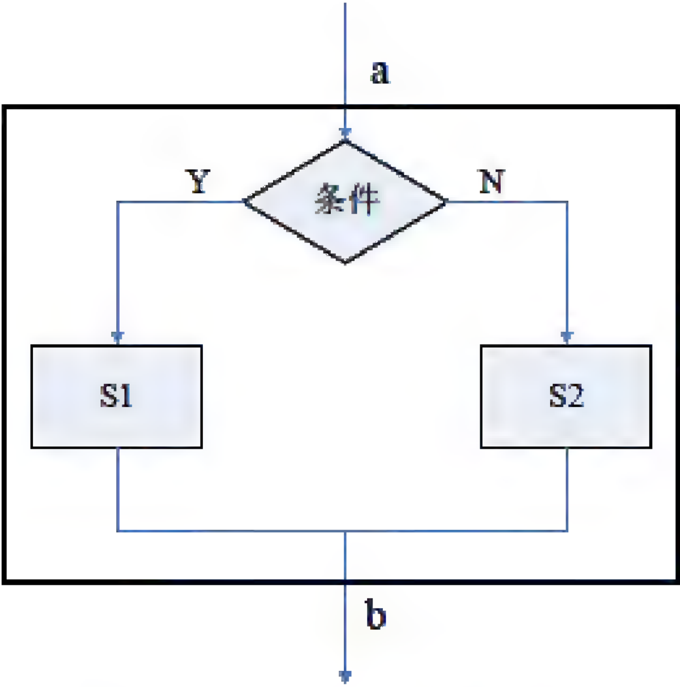


图 2.4　双选择结构

程序 2.5 给出了一个简单的选择结构程序的例子。

【程序 2.5】求两个数中的最大值：test5.c。

```
#include <stdio.h>
main()
{
        int x,y;
        printf("Enter x and y:\n");
        scanf("%d %d",&x,&y);
        if( x > y )
printf("max=%d\n",x);        /*如果条件满足执行此条语句*/
        else
 printf("max=%d\n",y);        /*如果条件不满足执行此条语句*/
}
```

程序运行结果如下(□表示空格，↙表示回车)：

```
Enter x and y:
10□5↙
max=10
```

多选择结构如图 2.5 所示，程序出现多个分支，程序执行方向将根据条件确定。如果满足条件 1 则执行 S1，如果满足条件 n 则执行 Sn。总之，要根据条件选择多个分支中的一个执行，不论选择哪一条分支，最后流程要到达同一个出口，如果所有分支条件都不满足，则直接到达出口。

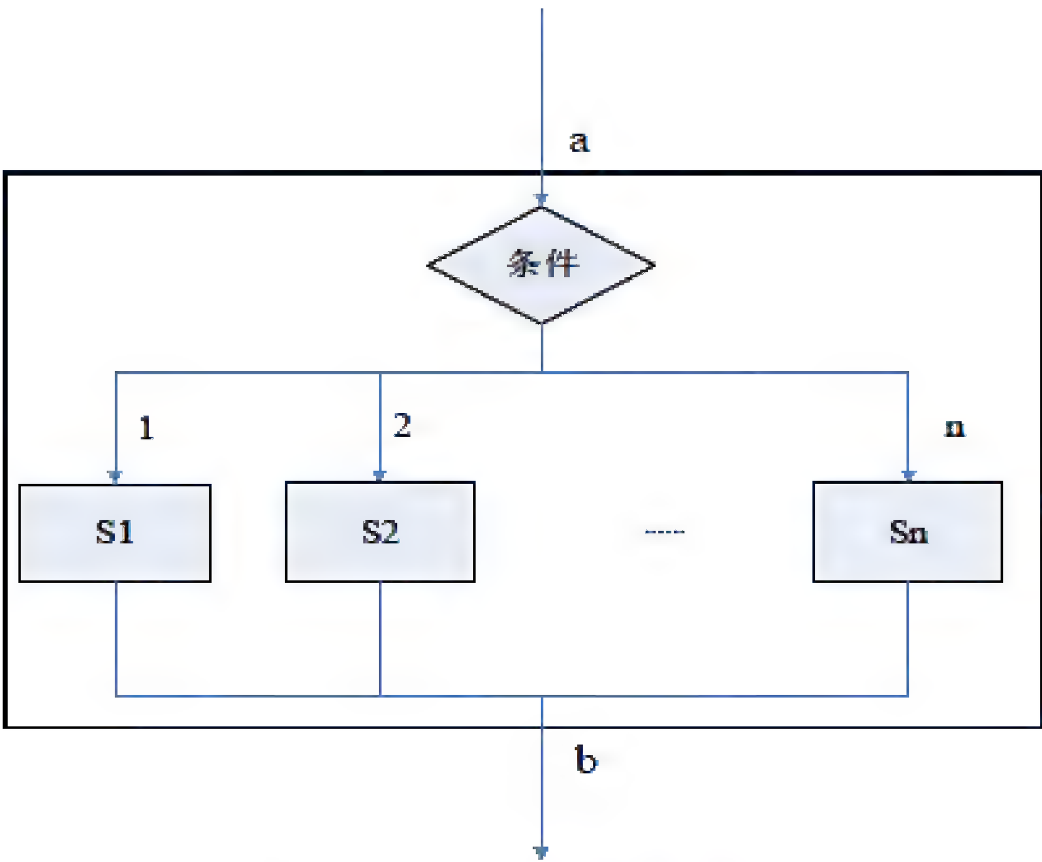


图 2.5　多选择结构

在 C 语言中，用嵌套 if 语句可以实现多分支结构程序，但分支较多时就显得很复杂，可读性差。C 语言中的 switch 语句专用于实现多分支结构程序。switch 语句的调用形式如下：

```
switch (表达式)
{
case 常量表达式 1: 语句 1;
case 常量表达式 2: 语句 2;
……
case 常量表达式 n: 语句 n;
default : 语句 n+1;
}
```

switch 语句的执行过程可描述为：首先计算表达式的值，然后依次与常量表达式 i(i=1，2，3，…，n)进行比较，若表达式的值与某常量表达式相等，则从该常量表达式处开始执行，直到 switch 语句结束。若所有的常量表达式 i(i=1，2，3，…，n)的值均不等于表达式的值，则从 default 处开始执行。程序 2.6 是一个关于多分支选择结构的例子。

【程序 2.6】输入某学生的成绩，输出该学生的成绩和等级(A 级：90～100，B 级：80～89，C 级：60～79，D 级：0～59)。

为了区分各分数段，将[0，100]每十分划分为一段，则 x/10 的值为 0 到 10，它们表示 11 个段：0～9 为 0 段，10～19 为 1 段，……，90～99 为第 9 段，100 为第 10 段，用 case 后的常量表示段号。例如，x=66，则 x/10 的值为 6，所以 x 在 6 段，即 60≤x<70，属于 C 级。若 x 不在[0，10]段内，则表示 x 是非法成绩，在 default 分支中处理。控制流程如图 2.6 所示，其代码实现如 test6.c。

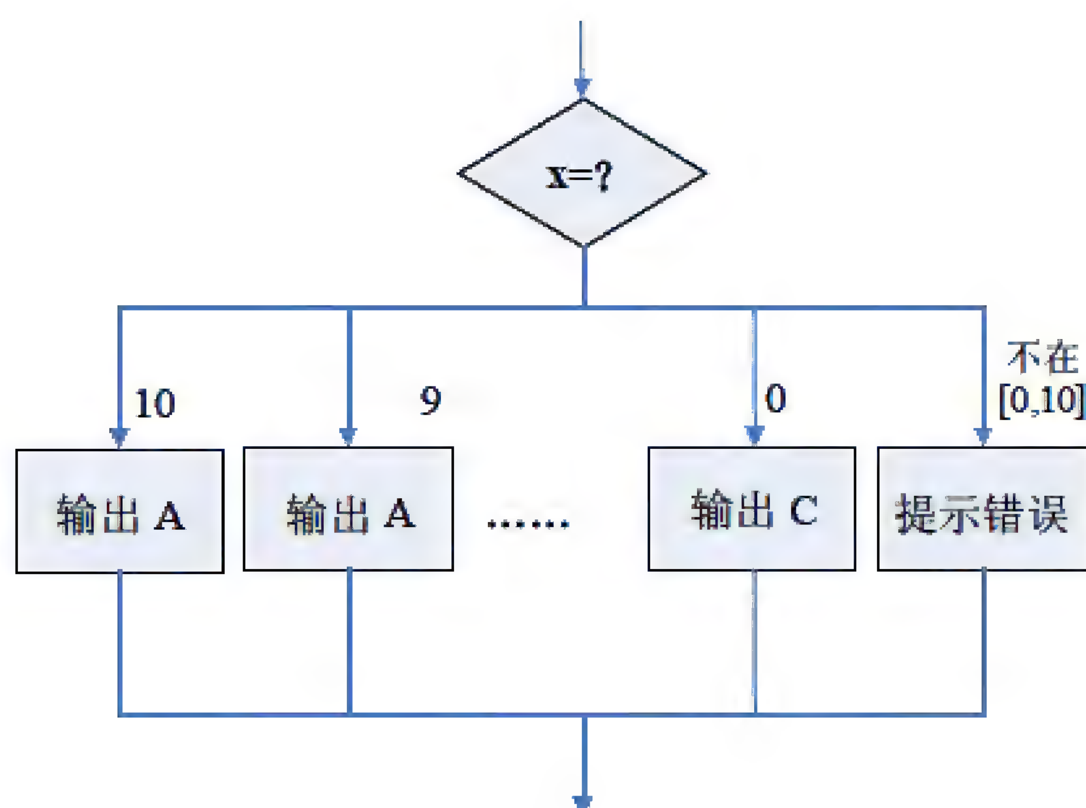


图 2.6　多分支结构

test6.c：

```
#include <stdio.h>
main()
{
        int x;
printf("Please input x:\n");
scanf("%d",&x);                          /*输入学生成绩*/
switch(x/10)                             /*判断条件*/
{
          case 10: printf("x=%d -> A\n",x);break;    /*分段显示结果*/
```

```
            case 9: printf("x=%d -> A\n",x);break;
            case 8: printf("x=%d -> B\n",x);break;
            case 7: printf("x=%d -> C\n",x);break;
            case 6: printf("x=%d -> C\n",x);break;
            case 5: printf("x=%d -> D\n",x);break;
            case 4: printf("x=%d -> D\n",x);break;
            case 3: printf("x=%d -> D\n",x);break;
            case 2: printf("x=%d -> D\n",x);break;
            case 1: printf("x=%d -> D\n",x);break;
            default: printf("x=%d data error!\n",x);     /*如果 x 不在[0，10]段内，则出错*/
        }
}
```

程序运行结果如下(↙表示回车)：

```
Please input x:
65↙
x=65 -> C
```

可以看到，当输入的成绩为 65 时，我们得到了等级 C。

说明

break 语句用于终止它所在的 switch 语句的执行。如果没有 break 语句，本例在执行完 case 6 后，还会依次执行 case 5、case 4 等后面的语句。读者可以自己验证一下不带 break 语句的情况。一般在多分支选择结构的程序中，在得到正确结果后，立即使用 break 语句来终止 switch 语句的执行。

2.5.3 循环结构

循环结构表示程序反复执行某个或某些操作，直到某些条件为假时才可终止循环。循环结构可以减少源程序重复书写的工作量，用来描述重复执行某段算法的问题，这是程序设计中最能发挥计算机特长的程序结构。

循环结构的基本形式有两种：当型循环和直到型循环，其流程分别如图 2.7(a)、(b)所示。

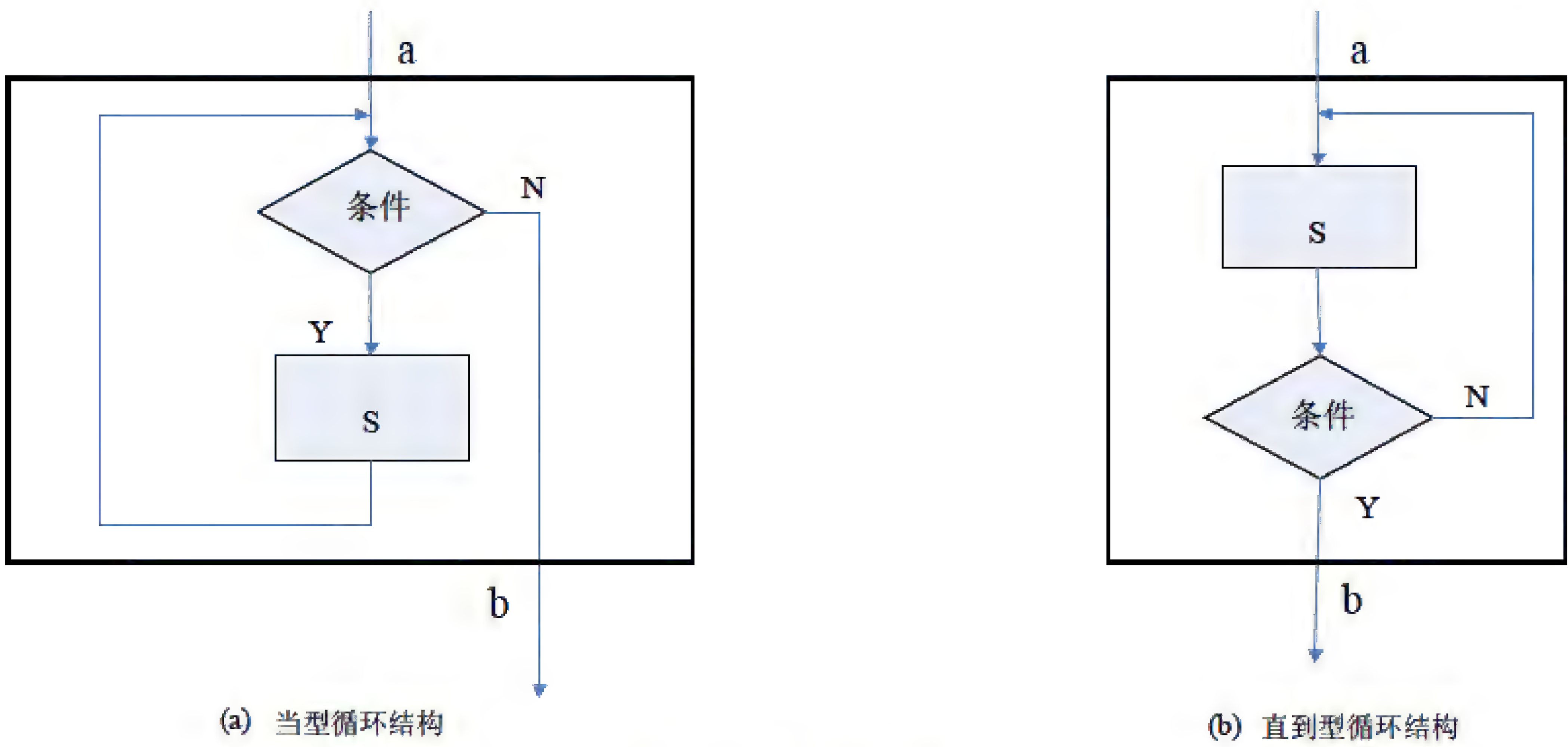


图 2.7　循环结构

当型循环结构的执行过程是，首先判断条件，当满足条件时执行循环体，执行完后自动返回循环入口；如果条件不满足，则退出循环体直接到达流程出口处，所以是先判断后执行。C 语言中用 while 语句实现当型循环结构。while 语句的调用形式为：

```
while (表达式) 循环体语句
```

直到型循环结构的执行过程是，从结构入口直接执行循环体，在循环终点处判断条件，如果条件不满足，返回入口处继续执行循环体，直到循环判断条件为真时再退出循环到达流程出口处，属于先执行后判断。C 语言用 do while 语句来实现直到型循环结构。do while 语句的调用形式为：

```
do
  循环体语句
while(表达式)
```

程序 2.7 是一个使用 do while 语句来实现直到型循环结构的例子。

【程序 2.7】求 n！(n！=1*2*3*…*(n-1)*n)。

这是若干项的连乘问题，执行流程如图 2.8 所示，代码实现如 test7.c。

test7.c：

```
#include <stdio.h>
main()
{
        int i,n;
long s;
        s=1;
        i=1;
        printf("Please input n: ");
        scanf("%d",&n);
        do{s*=i;          /*循环体*/
           i++;
        } while ( i<=n );    /*循环结束的判断条件*/
        printf("%d!=%d\n",n,s);
}
```

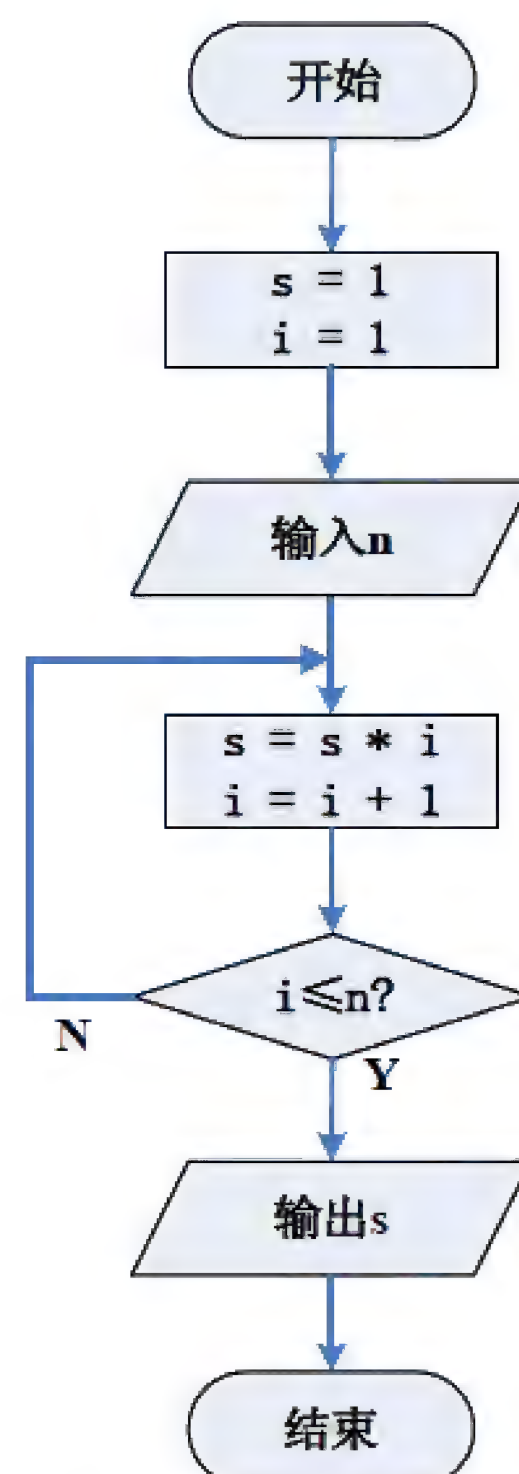


图 2.8　用 do_while 循环求 n！

程序运行结果如下(↙表示回车)：

```
Please input n: 5↙
5!=120
```

C 语言中，还有一种 for 循环语句。for 语句的调用形式如下：

```
for(表达式 1; 表达式 2; 表达式 3)
{循环体语句}
```

它等价于下列 while 语句：

```
表达式 1;
while(表达式 2){
```

```
    循环体语句;
    表达式 3;
}
```

for 语句的执行过程如图 2.9 所示。首先计算表达式 1 的值，然后检测表达式 2 的值，若其值为“真”，则执行循环体语句，执行完毕后，再计算表达式 3，然后检测表达式 2 的值，若为“真”，则继续执行循环体语句，如此循环，直到表达式 2 的值为“假”时终止循环。

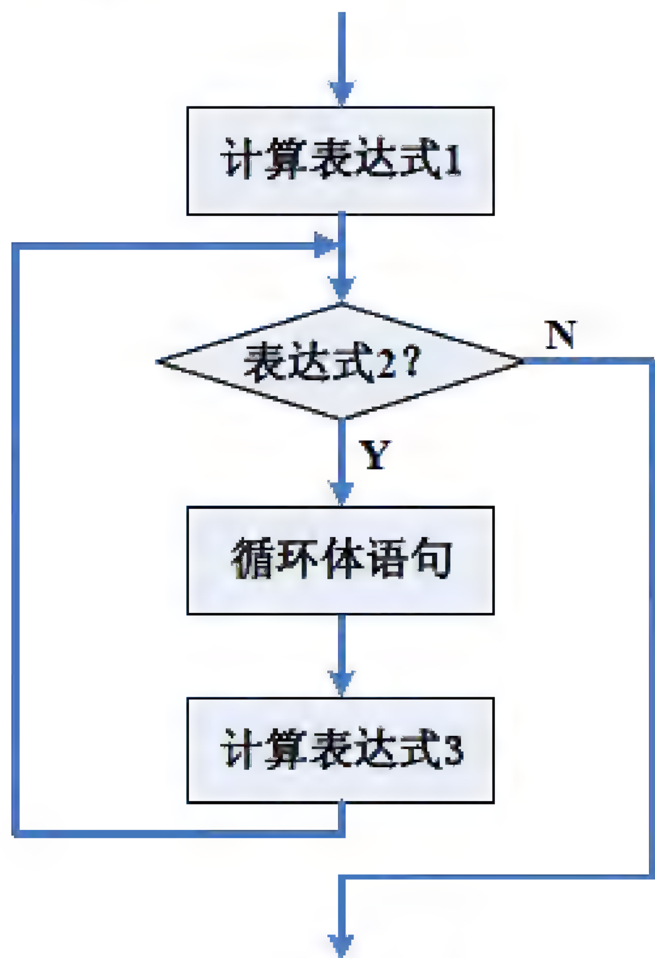


图 2.9　for 循环执行流程

表达式 1 通常是为循环变量赋初值的表达式，表达式 2 为控制循环的表达式，表达式 3 通常是改变循环变量的表达式。程序 2.8 给出了一个使用 for 语句的简单例子。

【程序 2.8】用 for 循环计算 1～100 的整数累加和：test8.c。

```
#include <stdio.h>
main()
{
        int i,n=0;
        for(i=1;i<=100;i++)
        {
            n+=i;
        }
        printf("n=%d",n);
}
```

程序运行结果如下：

```
n=5050
```

根据 for 循环的判断表达式 2，循环体一共执行了 100 次，即“n+=i;”语句一共被执行了 100 次。

2.6 C 语言中的数据输入与输出

C 语言中没有输入输出语句，为了实现输入输出功能，在 C 语言的库函数中提供了一组输

入输出函数，其中 scanf 和 printf 函数是针对标准输入输出设备(键盘和显示器)进行格式化输入输出的函数，而 getchar 和 putchar 是专门对单个字符进行输入输出的函数。要使用它们，必须将文件“stdio.h”包含在程序文件中。

2.6.1 字符输出函数 putchar

putchar 函数是字符输出函数，其功能是在终端(显示器)输出单个字符。其一般调用形式为：

```
putchar(字符变量);
```

例如：

```
putchar('A');          /*输出大写字母 A*/
putchar(x);            /*输出字符变量 x 的值*/
putchar('101');        /*也是输出字符 A*/
putchar('\n');         /*换行*/
```

注意，对控制字符则执行控制功能，不在屏幕上显示。

2.6.2 字符输入函数 getchar

getchar 函数的功能是从键盘上读一个字符作为函数值。其一般调用形式为：

```
getchar();
```

通常把输入的字符赋予一个字符变量，构成赋值语句，如程序 2.9 中的代码。

【程序 2.9】输入输出单个字符：test9.c。

```
#include <stdio.h>
void main()
{
    char c;                          /*定义字符变量 c*/
    printf("input a character:");    /*打印提示信息*/
    c=getchar();                     /*将读取的字符赋给变量 c*/
    putchar(c);                      /*输出字符变量 c*/
}
```

程序运行结果如下(↙表示回车)：

```
input a character:b↙
b
```

2.6.3 格式输出函数 printf

printf 函数称为格式输出函数。其功能是按照用户指定的格式，把指定的数据输出到显示器屏幕上。printf 函数调用的一般形式为：

```
printf("格式控制字符串", 输出项表);
```

其中格式控制字符串用来说明输出列表中各输出项的输出格式。输出项表列出了要输出的项，各输出项之间用逗号分开。输出项表也可以没有，这时输出的是格式字符串本身。

格式控制字符串有两种：格式字符串和非格式字符串。非格式字符串在输出时原样打印，在显示中只起提示作用。格式字符串是以“%”开头的字符串，在“%”后面跟有各种格式字符，以说明输出数据的类型、形式、长度、小数位数等。格式字符串的形式为：

```
% [输出最小宽度][.精度][长度]类型
```

如：%d，%9.3f 等。其中%d 格式符表示用十进制整型格式输出，而%f 表示用实型格式输出，附加格式说明符“9.3”表示输出宽度为 9(包括小数点)，并含 3 位小数。printf 函数常用的输出格式符及其含义如表 2.5 所示。

表 2.5　输出格式符

格 式 字 符	含　　义
d，i	以十进制形式输出带符号整数(正数不输出符号)
o	以八进制形式输出无符号整数(不输出前缀 0)
x	以十六进制形式输出无符号整数(不输出前缀 0x)
u	以十进制形式输出无符号整数
f	以小数形式输出单、双精度实数
e	以指数形式输出单、双精度实数
g	以%f 或%e 中较短输出宽度的一种格式输出单、双精度实数
c	输出单个字符
s	输出字符串

【程序 2.10】printf 函数输出整型、实型、字符型数据：test10.c。

```
#include <stdio.h>
main()
{
    int a=16;
    float b=123.4567;
    char c='A';
    printf("a=%d\n",a);              /*输出整型变量 a 的值*/
    printf("b=%9.4f\n",b);           /*输出实型变量 b 的值，注意运行结果的格式*/
    printf("c=%c,%s\n",c, "China"); /*输出字符变量 c 和字符串*/
}
```

程序运行结果如下(□表示空格)：

```
a=16
b=□123.4567
c=A,China
```

在程序 2.10 中，第一次用%d 格式输出整型数；第二次用%9.4f 格式输出实型数，宽度为 9(包括小数点)，并含 4 位小数，不足 9 列，则左端补空格；第三次是用%c 格式输出单个字符，用%s 格式输出字符串。

2.6.4 格式输入函数 scanf

scanf 函数称为格式输入函数，即按照格式字符串规定的格式，从键盘上把数据输入到指定的变量之中。scanf 函数调用的一般形式为：

```
scanf("格式控制字符串", 输入项地址表);
```

其中，格式控制字符串的作用与 printf 函数相同，但不能显示非格式字符串，也就是不能显示提示字符串。地址表项中给出各变量的地址，地址是由地址运算符“&”后跟变量名组成的(如&a, &b)。

scanf 函数中格式字符串的构成与 printf 函数基本相同，但使用时有几点不同。

(1) 格式说明符中，可以指定数据的宽度，但不能指定数据的精度。例如：

```
float a;
scanf("%10f",&a);          /*正确*/
scanf("%10.2f",&a);        /*错误*/
```

(2) 输入 long 型数据必须使用%ld，输入 double 数据必须使用%lf 或%le。

(3) 附加格式说明符“*”使对应的输入数据不赋给相应的变量，如程序 2.11。

【程序 2.11】用 scanf 函数读取输入的变量，并检查读取结果：test11.c。

```
#include <stdio.h>
main()
{
int a;
float b;
char c;
float d;                  /*定义 4 个不同数据类型的变量*/
scanf("%d",&a);           /*把输入的十进制整数赋给整型变量*/
printf("a=%d\n",a);
scanf("%10f",&b);         /*把输入的实数赋给实型变量*/
printf("b=%f\n",b);
scanf("%s",&c);           /*把输入的字符赋给字符型变量*/
printf("c=%c\n",c);
scanf("%*d",&d);          /*输入数据，不赋给相应的变量*/
printf("d=%f\n",d);
}
```

程序运行结果如下(↙表示回车)：

```
654↙
a=654
1.23↙
b=1.230000
e↙
c=e
4.6↙
d=-107374176.000000
```

2.7 函数

C 源程序是由函数组成的。最简单的程序有一个主函数 main()，但实用程序往往由多个函数组成，由主函数调用其他函数，其他函数也可以互相调用。函数是 C 源程序的基本模块，程序的许多功能是通过对函数模块的调用来实现的，学会编写和调用函数可以提高编程效率。

2.7.1 函数的定义

函数的定义通常包含以下内容：

```
类型  函数名(形参表说明)        /*函数首部*/
{
说明语句                    /*函数体*/
   执行语句
}
```

对上面的定义形式进行以下说明：

(1) “类型”是指函数返回值的类型。函数返回值不能是数组，也不能是函数，除此之外任何合法的数据类型都可以是函数的类型，如：int，long，float，char 等。函数类型可以省略，当不指明函数类型时，系统默认的是整型。

(2) 函数名是用户自定义的标识符，在 C 语言函数定义中不可省略，须符合 C 语言对标识符的规范，用于标识函数，并用该标识符调用函数。另外函数名本身也有值，它代表了该函数的入口地址，使用指针调用函数时，将用到此功能。

(3) 形参又称为“形式参数”。形参表是用逗号分隔的一组变量说明，包括形参的类型和形参的标识符，其作用是指出每一个形参的类型和形参的名称，当调用函数时，接收来自主调函数的数据，确定各参数的值。

(4) 用{ }括起来的部分是函数的主体，称为函数体。函数体是一段程序，确定该函数应完成的规定的运算，应执行的规定的动作，集中体现了函数的功能。函数内部应有自己的说明语句和执行语句，但函数内定义的变量不可以与形参同名。花括号{ }是不可以省略的。

根据函数定义的一般形式，可以定义一个最简单的函数：

```
add()
{  ;
}
```

这是 C 语言中一个合法的函数，函数名为 add。它没有函数类型说明，也没有形参表，同时函数体内也没有语句。实际上函数 add 不执行任何操作和运算，它是一个空函数，在一般情况下是没有用途的，但在程序开发的过程中有时是需要的，常用来代替尚未开发完毕的函数。

2.7.2 函数的调用

主调函数使用被调函数的功能，称为函数调用。在 C 语言中，只有在函数调用时，函数体

中定义的功能才会被执行。C 语言中，函数调用的一般形式为：

```
函数名(类型 形参，类型 形参…)；
```

对无参函数调用时则无实际参数表。实际参数表中的参数可以是常数、变量或其他构造类型数据及表达式，各实参之间用逗号分隔。

在 C 语言中，可以用以下几种方式调用函数。

(1) 函数表达式：函数作为表达式中的一项出现在表达式中，以函数返回值参与表达式的运算。这种方式要求函数是有返回值的。例如：

```
z=max(x,y);
```

是一个赋值表达式，把 max 的返回值赋予变量 z。

(2) 函数语句：函数调用的一般形式加上分号即构成函数语句。例如：

```
printf ("%d",a);
scanf ("%d",&b);
```

都是以函数语句的方式调用函数。

(3) 函数实参：函数作为另一个函数调用的实际参数出现。这种情况是把该函数的返回值作为实参进行传送，因此要求该函数必须是有返回值的。例如：

```
printf("%d",max(x,y));     /*把 max 调用的返回值作为 printf 函数的实参*/
```

在主调函数中调用某函数之前应对该被调函数进行声明。在主调函数中对被调函数进行说明的目的是使编译系统知道被调函数返回值的类型，以便在主调函数中按此种类型对返回值进行相应的处理。其一般形式为：

```
类型说明符 被调函数名(类型 形参，类型 形参…);
```

需要注意的是，函数的声明和函数的定义有本质上的不同。主要区别在以下两个方面：

(1) 函数的定义是编写一段程序，应有函数的具体功能语句——函数体；而函数的声明仅是向编译系统的一个说明，不含具体的执行动作。

(2) 在程序中，函数的定义只能有一次，而函数的声明可以有多次。

通过程序 2.12 中的代码，读者可以简单了解函数定义、说明、调用的全过程。

【程序 2.12】编写一个 max 函数，实现选取两个数中较大值的功能：test12.c。

```
#include <stdio.h>
int max(int a, int b);                    /*函数 max 的说明*/
main()
{
int a, b;
printf("Enter a b:");
scanf("%d %d",&a, &b);
    printf("max = %d\n", max(a,b)); /*函数 max 的调用*/
}
```

```
int max(int a, int b)              /*函数 max 的定义*/
{
int p;
        p = a>b?a:b;
        return (p);
}
```

程序运行结果如下(↙表示回车，□表示空格)：

```
Enter a b:100□99↙
max = 100
```

可以看到，程序中对两个输入整数的比较调用了 max()函数。

注 意

在下列情况下可以省去主调函数中对被调函数的函数说明。

(1) 如果被调函数的返回值是整型或字符型，可以不对被调函数进行说明，而直接调用。这时系统将自动对被调函数返回值按整型处理。

(2) 当被调函数的函数定义出现在主调函数之前时，在主调函数中也可以不对被调函数再进行说明而直接调用。

(3) 如在所有函数定义之前，在函数外预先说明了各个函数的类型，则在以后的各主调函数中，可不再对被调函数进行说明。

(4) 对库函数的调用不需要再进行说明，但必须把该函数的头文件用 include 命令包含在源文件前部。

2.7.3 变量的存储类别

在 C 语言中，变量是对程序中数据所占内存空间的一种抽象定义，定义变量时，用户定义变量的名、变量的类型，这些都是变量的操作属性。不仅可以通过变量名访问该变量，系统还通过该标识符确定变量在内存中的位置。在计算机中，保存变量当前值的存储单元有两类，一类是内存，另一类是 CPU 的寄存器。变量的存储类型关系到变量的存储位置，C 语言中定义了 4 种存储属性，即自动变量、外部变量、静态变量和寄存器变量，它关系到变量在内存中的存放位置，由此决定了变量的保留时间和变量的作用范围。

变量的保留时间又称为生存期，从时间的角度，可将变量分为静态存储和动态存储两种情况。静态存储是指变量存储在内存的静态存储区，在编译时就分配了存储空间，在整个程序的运行期间，该变量占有固定的存储单元，程序结束后，这部分空间才释放，变量的值在整个程序中始终存在；动态存储是指变量存储在内存的动态存储区，在程序的运行过程中，只有当变量所在的函数被调用时，编译系统才临时为该变量分配一段内存单元，函数调用结束，该变量空间释放，变量的值只在函数调用期存在。

变量的作用范围又称为作用域，从空间角度来分，可以分为全局变量和局部变量。局部变量是在一个函数或复合语句内定义的变量，它仅在函数或复合语句内有效，编译时，编译系统不为局部变量分配内存单元，而是在程序运行过程中，当局部变量所在的函数被调用时，编译

系统根据需要，临时分配内存，调用结束，空间释放；全局变量是在函数之外定义的变量，其作用范围为从定义处开始到本文件结束，编译时，编译系统为其分配固定的内存单元，在程序运行的自始至终都占用固定单元。

1. 自动变量

函数中的局部变量，如不专门声明为 static 存储类别，都是动态地分配存储空间的，数据存储在动态存储区中。函数中的形参和在函数中定义的变量(包括在复合语句中定义的变量)都属此类，在调用该函数时系统会给它们分配存储空间，在函数调用结束时就自动释放这些存储空间。这类局部变量称为自动变量。自动变量用关键字 auto 进行存储类别的声明，例如声明一个自动变量：

```
int fun(int a)
{
auto int b,c=3;        /*定义 b，c 为自动变量*/
}
```

a 是函数 fun()的形参，b、c 是自动变量，并对 c 赋初值 3。执行完 fun()函数后，自动释放 a、b、c 所占的存储单元。

2. 外部变量

外部变量(即全局变量)是在函数的外部定义的，它的作用域为从变量定义处开始，到本程序文件的末尾。如果外部变量不在文件的开头定义，其有效的作用范围只限于定义处到文件末尾。如果在定义点之前的函数想引用该外部变量，则应该在引用之前用关键字 extern 对该变量进行“外部变量声明”。表示该变量是一个已经定义的外部变量。有了此声明，就可以从“声明”处起，合法地使用该外部变量，如程序 2.13 所示。

【程序 2.13】用 extern 声明外部变量，扩展程序文件中的作用域：test13.c。

```
#include <stdio.h>
int max(int x,int y)            /*定义 max 函数*/
{
int z;
    z=x>y?x:y;
    return(z);
}
main( )
{
extern A,B;                     /*声明外部变量*/
    printf("%d\n",max(A,B));    /*调用 max 函数*/
}
int A=13,B=-8;                  /*在文件末尾定义外部变量*/
```

在本程序文件的最后 1 行定义了外部变量 A、B，但由于外部变量定义的位置在主函数 main 之后，因此本来在 main 函数中不能引用外部变量 A、B。但由于在 main 函数中用 extern 对 A 和 B 进行了“外部变量声明”，就可以从“声明”处起，合法地使用该外部变量 A 和 B 了。

3. 静态变量

有时希望函数中的局部变量的值在函数调用结束后不消失而保留原值，这时就应该指定局部变量为静态局部变量，用关键字 static 进行声明。考察静态局部变量的值，如程序 2.14 所示。

【程序 2.14】静态局部变量的使用：test14.c。

```
#include <stdio.h>
fun(int a)                /*定义函数 fun*/
{
auto b=0;            /*定义自动变量 b，并初始化为 0*/
    static c=3;            /*定义静态变量 c，并初始化为 3*/
    b=b+1;
    c=c+1;
    return(a+b+c);
}
main()
{
int a=2,i;
    for(i=0;i<3;i++)     /*重复调用函数 f，比较函数返回值*/
      printf("%d ",fun(a));
}
```

程序运行结果如下(□表示空格)：

```
7□8□9
```

在上面的程序中，对于自动变量 b，每次调用函数 fun 时，其值都被初始化为 0，而静态变量 c 则保留上一次调用结束时的值，即逐次加 1，所以程序运行的结果也是逐次增加 1。

提 示

从静态可以看到，当输入的成绩为 65 时，我们得到了等级 C 局部变量与自动变量的不同：

(1) 静态局部变量属于静态存储类别，在静态存储区内分配存储单元。在整个程序运行期间都不释放。而自动变量(即动态局部变量)属于动态存储类别，占动态存储空间，函数调用结束后即释放。

(2) 静态局部变量在编译时赋初值，即只赋初值一次；而对自动变量赋初值是在函数调用时进行，每调用一次函数重新给一次初值，相当于执行一次赋值语句。

(3) 如果在定义局部变量时不赋初值的话，则对静态局部变量来说，编译时自动赋初值 0(对数值型变量)或空字符(对字符变量)。而对自动变量来说，如果不赋初值则它的值是一个不确定的值。

4. 寄存器变量

为提高效率，C 语言允许将局部变量的值存放在 CPU 的寄存器中，这种变量叫作寄存器变量，用关键字 register 声明。使用寄存器变量需要注意以下几点：

(1) 只有局部自动变量和形式参数可以作为寄存器变量。

(2) 一个计算机系统中的寄存器数目有限，不能定义任意多个寄存器变量。

(3) 不能使用取地址运算符“&”求寄存器变量的地址。

【程序 2.15】定义一个函数，实现阶乘功能，函数中使用寄存器变量：test15.c。

```
#include <stdio.h>
int fac(int n)              /*定义阶乘函数*/
{
        register int i,f=1;  /*定义寄存器变量 i,f*/
        for(i=1;i<=n;i++)   /*用循环实现阶乘的功能*/
          f=f*i;
        return(f);          /*返回计算结果 f*/
}
main()
{
        int i;
        for(i=0;i<=5;i++)
          printf("%d! = %d\n",i,fac(i)); /*调用阶乘函数*/
}
```

程序运行结果如下：

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

最后，表 2.6 对这 4 种变量存储类别的特性进行了总结。

表 2.6　4 种存储类型的特性

性　　能	自动变量	外部变量	静态变量		寄存器变量
			外部	内部	
记忆能力	无	有	有	有	无
多个函数共享	否	是	是	否	否
整个程序的不同文件共享性	否	是	否	否	否
初始化时未显示赋值的取值	不确定	0	0	0	不确定
变量初始化	由程序控制	编译器	编译器	编译器	由程序控制
数组与结构初始化	是	是	是	是	否
作用域	当前函数	整个程序	文件	函数	当前函数

2.8 数组

数组是同类型有序数据的集合，可以为这些数据的集合起一个名字，称为数组名。该集合

中的各个数据项称为数组元素，每个元素可用数组名和下标表示。在 C 程序设计中，数组是一个十分有用的数据类型，下面将对数组进行详细介绍。

2.8.1 一维数组的定义和使用

在 C 语言中使用数组必须先进行定义，一维数组的定义方式如下：

```
类型说明符 数组名 [常量表达式];
```

其中类型说明符是任意一种基本数据类型或构造数据类型，它定义了全体数组成员的数据类型；数组名是用户定义的数组标识符；方括号中的常量表达式表示数据元素的个数，也称为数组的长度。例如：

```
float a[5],b[10];
```

该语句表示：

(1) 定义了浮点型数组 a 和 b，其数组元素的类型都是 float。

(2) a 数组有 5 个数组元素，b 数组有 10 个元素。

(3) a 数组的数组元素是 a[0]、a[1]、a[2]、a[3]和 a[4]，共 5 个数组元素。所以 a 数组元素的下标大于等于 0，且小于 5。

(4) 定义了 float 型数组 a，编译程序将为 a 数组在内存中开辟 5 个连续的存储单元，用来存放 a 数组的 5 个数组元素，a[0]代表这片存储区的第一个存储单元。数组名 a 代表 a 数组的首地址，即 a[0]的地址。

数组元素是组成数组的基本单元，数组元素也是一种变量，其标识方法为数组名后跟一个下标。下标表示元素在数组中的顺序号。引用数组元素的一般形式为：

```
数组名[下标]
```

其中下标只能为整型常量或整型表达式。例如 a[5]、a[i+j]、a[i++]都是合法的数组元素。

数组元素通常也称为下标变量。必须先定义数组，才能使用下标变量。在 C 语言中只能逐个使用下标变量，而不能一次引用整个数组。

给数组赋值的方法除了用赋值语句对数组元素逐个赋值外，还可采用初始化赋值和动态赋值的方法。数组初始化赋值是指在数组定义时给数组元素赋予初值。数组初始化是在编译阶段进行的，这样将减少程序运行时间，提高效率。

初始化赋值的一般形式为：

```
类型说明符 数组名[常量表达式]={初始值, 初始值, …… 初始值};
```

例如：

```
int a[10]={ 0,1,2,3,4,5,6,7,8,9 };
```

相当于：

```
a[0]=0;a[1]=1...a[9]=9;
```

在输出数组时，通常使用循环语句逐个输出各下标变量。程序 2.16 是关于数组初始化与

输出的简单例子。

【程序 2.16】定义一个数组，逐个对其赋值，然后输出各个元素值：test16.c。

```
#include <stdio.h>
main()
{
    int i,a[10];            /*定义数组 a*/
    for(i=0;i<=9;i++)       /*使用 for 循环依次对数组中的各个元素赋初值*/
        a[i]=i;
    for(i=9;i>=0;i--)       /*使用 for 循环依次输出数组的每个元素*/
        printf("%d ",a[i]);
}
```

程序运行结果如下(□表示空格)：

```
9□8□7□6□5□4□3□2□1□0
```

程序 2.16 中首先使用 for 循环依次对数组 a 中的各个元素赋初值，再用 for 循环依次输出数组的各个元素值。

2.8.2　二维数组的定义和使用

当数组元素具有两个下标时，该数组称为二维数组，同样地，n 维数组具有 n 个下标。在实际问题中有很多量是二维的或多维的，多维数组元素有多个下标，以标识它在数组中的位置，所以也称为多下标变量。二维数组定义的一般形式是：

```
类型说明符 数组名[常量表达式 1][常量表达式 2];
```

其中常量表达式 1 表示第一维下标的长度，常量表达式 2 表示第二维下标的长度。

例如：

```
int a[2][3];
```

该语句表示：

(1) 定义了整型二维数组 a，其数组元素类型是 int。

(2) a 数组有两行三列，共 6 个元素。

(3) 该数组的行下标为 0、1，列下标为 0、1、2。数组元素分别是：

```
a[0][0],a[0][1],a[0][2],a[1][0],a[1][1],a[1][2];
```

(4) 定义了 int 型数组 a，编译程序将为 a 数组在内存中开辟 6 个连续的存储单元，用来存放 a 数组的 6 个元素。存储方式为按行存放，即先依次存放第 0 行的 3 个元素，然后再接着存放第 1 行的 3 个数组元素。数组名 a 代表数组的首地址。

(5) 在 C 语言中，二维数组 a 的每一行都可以看作是一维数组，a[0]表示第 0 行的 3 个元素构成的一维数组。

同一维数组一样，引用二维数组，也是引用它的数组元素。其表示的形式为：

```
数组名[行下标][列下标]
```

其中下标应为整型常量或整型表达式。

二维数组初始化也是在类型说明时给各下标变量赋予初值，二维数组可按行分段赋值，也可按行连续赋值。例如对数组 a[4][3]。

(1) 按行分段赋值可写为：

```
int a[4][3]={ {80,75,92},{61,65,71},{59,63,70},{85,87,90} };
```

(2) 按行连续赋值可写为：

```
int a[4][3]={ 80,75,92,61,65,71,59,63,70,85,87,90};
```

这两种赋初值的结果是完全相同的。程序 2.17 是二维数组定义与使用的简单例子。

【程序 2.17】一个学习小组共有 5 人，每个人有三门课的考试成绩，求各科的平均成绩和全组成员的总平均成绩。他们的成绩与科目如下表所示：

	张	王	李	赵	周
Math	80	61	59	85	76
C	75	65	63	87	77
Foxpro	92	71	70	90	85

test17.c：

```
#include <stdio.h>
main()
{
      int i,j,s=0, average,v[3];
      int a[5][3]={{80,75,92},{61,65,71},{59,63,70},{85,87,90},{76,77,85}};/*定义二维数组*/
      for(i=0;i<3;i++)        /*用两层循环嵌套的方式访问数组的每个元素*/
      {
      for(j=0;j<5;j++)
                  s=s+a[j][i];       /*变量 s 的值为各科的总成绩*/
            v[i]=s/5;              /*各科的平均成绩*/
            s=0;
      }
average=(v[0]+v[1]+v[2])/3;    /*总平均成绩*/
      printf("Math:%d\nC language:%d\nFoxpro:%d\n",v[0],v[1],v[2]);
      printf("total:%d\n", average);
}
```

程序运行结果如下：

```
Math:72
C language:73
Foxpro:81
total:75
```

2.8.3 字符数组和字符串

用来存放字符的数组称为字符数组。字符数组的各个元素依次存放字符串的各字符，字符

数组的数组名代表该数组的首地址，这为处理字符串中个别字符和引用整个字符串提供了极大的方便。

字符数组的定义形式与前面介绍的数值数组相同。例如：

```
char c[10];
```

字符数组也允许在定义时进行初始化赋值。例如：

```
char c[6]={'c', ' h ', 'i', 'n', 'a' , '\0' };
```

对字符数组的各个元素逐个赋值后，各元素的值为：

```
c[0]= 'c',c[1]= 'h',c[2]= 'i',c[3]= 'n',c[4]= 'a',c[5]= '\0'
```

其中，'\0'为字符串结束符。如果不对 c[5]赋任何值，'\0'会由系统自动添加。

字符数组也可采用字符串常量的赋值方式，例如：

```
char a[ ]={"china"};
```

2.8.4 常用字符串处理函数

C 语言提供了丰富的字符串处理函数，大致可分为字符串的输入、输出、合并、修改、比较、转换、复制、搜索等几类，使用这些函数可大大减轻编程的负担。用于输入输出的字符串函数，在使用前应包含头文件 stdio.h，使用其他字符串函数则应包含头文件 string.h。下面介绍几个最常用的字符串处理函数。

1. 字符串输出函数 puts

格式：

```
puts (字符数组名);
```

功能：把字符数组中的字符串输出到显示器，即在屏幕上显示该字符串，如程序 2.18 所示。

【程序 2.18】用 puts 函数输出一个字符串：test18.c。

```
#include <stdio.h>
main()
{
    char c[]="BASIC\nBASE";         /*定义一个字符串数组*/
    puts(c);                        /*输出字符串*/
}
```

程序运行结果如下：

```
BASIC
BASE
```

字符串数组中的“\n”为转义字符，意思为换行。

提 示

puts 函数完全可以由 printf 函数取代。当需要按一定格式输出时，通常使用 printf 函数。

2. 字符串输入函数 gets

格式：

```
gets(字符数组名);
```

功能：从标准输入设备上输入一个字符串，如程序 2.19 所示。

【程序 2.19】用 gets 函数读取一个字符串：test19.c。

```
#include <stdio.h>
main()
{
   char st[15];                    /*定义一个字符串数组*/
    printf("input string: ");
     gets(st);                      /*输入字符串*/
     puts(st);                      /*输出字符串*/
}
```

程序运行结果如下(□表示空格，↙表示回车)：

```
input string: abcde□fg↙
abcde□fg
```

提 示

gets 函数并不以空格作为字符串输入结束的标志，而是以回车作为输入结束的标志，这与 scanf 函数是不同的。

3. 字符串连接函数 strcat

格式：

```
strcat (字符数组名 1，字符数组名 2);
```

功能：把字符数组 2 中的字符串连接到字符数组 1 中字符串的后面，并删除字符串 1 后的结束标志‘\0’，函数的返回值是字符数组 1 的首地址。如程序 2.20 所示。

【程序 2.20】用 strcat 函数连接两个字符串：test20.c。

```
#include <string.h>                   /*字符串处理函数头文件*/
main()
{
     static char st1[30]="My name is ";    /*定义字符串数组 st1*/
     int st2[10];                         /*定义数组 st2 为整型*
     printf("input your name: ");
     gets(st2);                           /*输入字符串 st2*/
     strcat(st1,st2);                     /*将字符串 st2 连接到 st1 的后面*/
     puts(st1);                           /*输出字符串 st1*/
}
```

程序运行结果如下(↙表示回车)：

```
input your name: LiLei↙
```

```
My name is LiLei
```

从程序 2.20 中也可以看出，整型的字符串数组和字符型的字符串数组是可以相互赋值的，在 C 语言中，两者可以看作是等同的。

注 意

在使用 strcat 函数时，字符数组 1 应定义足够的长度，否则不能全部装入被连接的字符串。

4. 字符串复制函数 strcpy

格式：

```
strcpy (字符数组名 1, 字符数组名 2);
```

功能：把字符数组 2 中的字符串复制到字符数组 1 中，字符串结束标志‘\0’也一同复制。字符数组 2 也可以是一个字符串常量，这时相当于把一个字符串赋给一个字符数组。如程序 2.21 所示。

【程序 2.21】用 strcpy 函数将 str2 中的字符串复制到 str1 中：test21.c。

```
#include <string.h>                       /*字符串处理函数头文件*/
main()
{
    char st1[15],st2[]="C Language";      /*定义两个字符串数组*/
    strcpy(st1,st2);                      /*将 str2 中的字符串复制到 str1 中*/
    puts(st1);                            /*输出字符串 st1*/
}
```

程序运行结果如下：

```
C Language
```

注 意

同 strcat 函数一样，使用 strcpy 函数时，字符数组 1 也应定义足够的长度，否则不能全部装入所复制的字符串。

5. 字符串比较函数 strcmp

格式：

```
strcmp(字符数组名 1, 字符数组名 2);
```

功能：按 ASCII 码值的大小逐个比较两个字符串数组中的各个字符，直到出现不同的字符或遇到‘\0’为止。函数的返回值有以下 3 种情况：

- 字符串 1＝字符串 2，返回值为 0。
- 字符串 1＞字符串 2，返回值为一正整数。
- 字符串 1＜字符串 2，返回值为一负整数。

strcmp 函数也可用于比较两个字符串常量，或比较数组和字符串常量，如程序 2.22 所示。

【程序 2.22】比较两个字符串的大小：test22.c。

```
#include <string.h>                        /*字符串处理函数头文件*/
main()
{
 int k;
    static char st1[15],st2[]="abcd";   /*定义两个字符串数组*/
   printf("input a string: ");
   gets(st1);                          /*输入字符串 st1*/
   k=strcmp(st1,st2);                  /*比较字符串 st1 和 st2*/
   if(k==0) printf("st1=st2\n");       /*比较结果*/
   if(k>0) printf("st1>st2\n");
   if(k<0) printf("st1<st2\n");
}
```

程序运行结果如下(↙表示回车)：

```
input a string: abck↙
st1>st2
```

本程序中把输入的字符串和数组 st2 中的字符串比较，比较结果返回给变量 k，根据 k 值再输出比较结果。

6. 求字符串长度函数 strlen

格式：

```
strlen(字符数组名);
```

功能：求字符串的实际长度(不含字符串结束标志‘\0’)，并作为函数返回值，如程序 2.23 所示。

【程序 2.23】验证 strlen 函数的功能：test23.c。

```
#include <string.h>                        /*字符串处理函数头文件*/
main()
{
int k;
    static char str[]="abcde";
    k=strlen(str);             /*求字符串 str 的长度*/
    printf("The lenth of the string is %d\n",k);
}
```

程序运行结果如下：

```
The lenth of the string is 5
```

可以看到，求取字符串的长度时，是指字符串的实际长度，并不包含在内存中自动添加的字符串结束标识符‘\0’。

2.9 指针

指针是 C 语言中的一种数据类型。掌握指针型数据的使用，是深入理解 C 语言特性和掌

握 C 语言编程技巧的重要环节，正确灵活地使用指针，可以有效地描述各种复杂的数据结构，能够动态地分配内存空间，能够方便地操作字符串，还可以自由地在函数之间传递各种类型的数据，使程序简洁、紧凑，执行效率高。

2.9.1 地址和指针

首先需要了解程序中的数据在内存中是怎样进行存储的。在对程序进行编译时，程序中定义的变量会被分配到内存中的某一个单元，内存单元的长度由变量的类型决定。例如，int 型变量分配 2 个字节，float 型变量分配 4 个字节，char 型变量分配 1 个字节。C 程序中的变量在内存中占有一个内存单元，每个内存单元由若干个字节组成，每个字节都有自己的编号(即地址)，而一个变量的地址是指该变量的内存单元中第一个字节的编号。C 语言允许在程序中使用变量的地址，并可以通过地址运算符“&”得到变量的地址，例如：

```
float x;
int a[10];
```

通过&x 和数组名 a，就可以获得变量 x 和数组变量 a 的地址。

C 语言通过直接访问和间接访问两种方式来读取内存中的变量。直接访问是通过变量名或地址访问变量的存储区，例如：

```
scanf ("%d", &x );
x = sqrt(x);
printf ("%d", x );
```

间接访问是将一个变量的地址存放在另一个变量中。如图 2.10 所示，变量 x 的存储单元地址为 1010，将变量 x 的地址值存放在变量 p 中，访问 x 时先找到 p，再由 p 中存放的地址值找到 x。

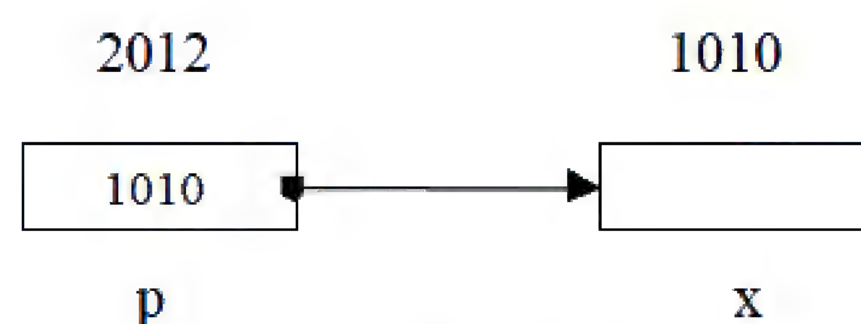


图 2.10 变量的间接访问

知道了数据在内存中的存储和读取方式后，指针的概念就不难理解了。一个变量的指针就是该变量的地址(指针就是地址)，如变量 x 的指针即为 1010。

指针变量就是指存放某个变量的地址的变量，它用来指向另一个变量，如图 2.10 所示中的 p。

2.9.2 指针的定义和使用

对指针变量定义的一般形式为：

```
类型说明符 *变量名;
```

其中，*表示这是一个指针变量，变量名即为定义的指针变量名，类型说明符表示该指针

变量所指向的变量的数据类型。例如：

```
int *p;     /*p 是指向整型变量的指针变量*/
```

说 明

该语句表示 p 是一个指针变量，它的值是某个整型变量的地址，或者说 p 指向一个整型变量。至于 p 究竟指向哪一个整型变量，应由向 p 赋予的地址来决定。

在使用指针变量时，要首先对指针变量赋初值，使指针变量指向一个具体值。为指针变量赋值的方式有两种，使用赋值语句为指针赋初值和在定义指针变量的同时进行初始化。例如：

```
int a, *pa;
pa = &a;            /*方式一：使用赋值语句为指针赋初值*/
int *pb = &a;       /*方式二：定义指针变量的同时进行初始化*/
```

在指针定义和使用的过程中，经常会用到“&”和“*”这两个运算符。“&”是取地址运算符，“*”为指针运算符。例如：

```
int x = 10, *p, y;
p = &x;             /*把变量 x 的地址赋给指针变量 p*/
y = * p;            /* *p 表示指针变量 p 所指单元的内容，即变量 x 的值，则 y=10 */
```

提 示

在这个例子中，虽然第一条语句和第三条语句都出现了“*p”，但它们的意义却不同，这是因为“*”在类型说明和取值运算中的含义是不同的，初学者要多加注意。

2.9.3 数组与指针

前面我们已经知道，通过数组下标可以确定数组元素在数组中的顺序和存储地址。由于每个数组元素相当于一个变量，因此指针变量可以指向数组中的元素，也就是说可以用指针方式访问数组中的元素。

对一个指向数组元素的指针变量的定义和赋值方法，与指针变量相同。例如：

```
int a[10];    /*定义 a 为包含 10 个整型数据的数组*/
int *p;       /*定义 p 为指向整型变量的指针*/
p=&a[0];      /*把 a[0]元素的地址赋给指针变量 p*/
```

C 语言规定，数组名代表数组的首地址，也就是第 0 号元素的地址。因此：

```
p=a;      /*等价于 p=&a[0]; */
int *p=a;     /*等价于 int *p=&a[0]; */
```

对于指向首地址的指针 p，p+i(或 a+i)就是数组元素 a[i]的地址，*(p+i)(或*(a+i))就是 a[i]的值。如果指针变量 p 已指向数组中的某一个元素，则 p+1 指向同一数组中的下一个元素。

引入指针变量后，就可以用以下两种方法来访问数组元素：

(1) 下标法，即用 a[i]形式访问数组元素，在前面介绍数组时都是采用这种方法。

(2) 指针法，即采用*(a+i)或*(p+i)形式，用间接访问的方法来访问数组元素，其中 a 是数组名，p 是指向数组的指针变量，其初值 p=a。具体实现过程如程序 2.24 中的代码。

【程序 2.24】定义一个数组，并用指针法访问数组的元素：test24.c

```
#include <stdio.h>
main()
{
int i;
int a[5]={0,1,2,3,4};               /*声明一个数组并对其进行初始化*/
int *p = a;                         /*把数组的首地址赋给指针变量 p*/
for(i=0; i<5; i++)
        printf("a[%d] = %d\n",i,*(a+i));   /*通过数组名计算元素的地址，找出元素的值*/
for(i=0; i<5; i++)
        printf("a[%d] = %d\n",i,*(p+i));   /*用指针变量指向元素*/
}
```

程序运行结果如下：

```
a[0]=0
a[1]=1
a[2]=2
a[3]=3
a[4]=4
a[0]=0
a[1]=1
a[2]=2
a[3]=3
a[4]=4
```

2.9.4 字符串与指针

前面我们已经讨论过字符数组与字符串，字符指针也可以指向一个字符串，可以用字符串常量对字符指针进行初始化。例如：

```
char *str = " This is a string." ;
```

这是对字符指针进行初始化。此时，字符指针指向一个字符串常量的首地址。

还可以用字符数组来存放字符串，例如：

```
char string[ ] = "This is a string.";
```

在这个语句中，string 是数组名，代表字符数组的首地址。因此可以通过数组名 string 来访问字符串。

字符串指针和字符串数组两种方式都可以访问字符串，但它们有着本质的区别：字符指针 str 是个变量，可以改变 str 使它指向不同的字符串，但不能改变 str 所指向的字符串常量的值。而 string 是一个数组，可以改变数组中保存的内容。读者应注意字符串指针和字符串数组的区别。

2.9.5 指向函数的指针

在定义一个函数之后，编译系统为每个函数确定一个入口地址，当调用该函数的时候，系统会从这个“入口地址”开始执行该函数。存放函数入口地址的变量就是一个指向函数的指针，简称为函数指针。函数指针定义的一般形式如下：

```
类型标识符 (* 指针变量名) ();
```

类型标识符为函数返回值的类型。在 C 语言中，()的优先级比*高，因此，“* 指针变量名”外部必须用括号，否则指针变量名首先与后面的()结合。

函数指针必须赋初值，才能指向具体的函数。由于函数名代表了该函数的入口地址，因此可以直接用函数名为函数指针变量赋初值，即：

```
函数指针变量名 = 函数名;
```

例如：

```
double fun();        /*函数说明*/
double (* f)();      /*函数指针说明*/
f = fun;             /*f 指向 fun 函数*/
```

函数指针经定义和赋值之后，在程序中可以应用该指针，目的是调用被指针所指的函数。由此可见，使用函数型指针，增加了函数调用的方式。

2.10 结构体和共用体

在实际生活中，有大量由不同性质的数据构成的实体，如通信录就是由姓名、地址、电话、邮编等信息组成的。对于这种实体，用数组是难以描述的。因此，C 语言提供了一种被称为结构体的构造型数据类型，结构类型为处理复杂数据类型提供了便利的手段。

2.10.1 定义和引用结构体

结构体与数组类似，都是由若干分量组成的，与数组不同的是，结构体的分量可以是不同类型，可以通过成员名来访问结构体的元素。

结构体的定义说明了它的组成成员，以及每个成员的数据类型。定义一般形式如下：

```
struct 结构类型名
{
数据类型    成员名 1;
数据类型    成员名 2;
……
数据类型    成员名 n;
};
```

例如，定义一个结构体 address，用它来记录通信录信息：

```
struct address
{
        char name[30];      /*姓名，字符数组作为结构体中的成员 */
        char street[40];    /*街道*/
        unsigned long tel;  /*电话，无符号长整型作为结构体中的成员 */
        unsigned long zip;  /*邮政编码*/
}
```

结构的定义说明了变量在结构中的存在格式，要使用该结构就必须说明结构类型的变量。结构变量说明的一般形式如下：

```
struct 结构类型名称 结构变量名;
```

定义结构体便是定义了一种由成员组成的复合类型，而用这种类型说明了一个变量才会产生具体的实体。与说明基本数据类型的变量一样，系统会按照结构定义时的内部组成，为说明的结构变量分配内存空间。结构变量的成员在内存中占用连续的存储区域，所占内存大小为结构中每个成员的长度之和。

我们可以将变量 student1 说明为 address 类型的结构变量：

```
struct address student1;
```

虽然，结构体作为若干成员的集合是一个整体，但在使用结构时，不仅要对结构的整体进行操作，而且还经常要访问结构中的每一个成员。在程序中使用机构中成员的方法为：

```
结构变量名.成员名称
```

如 student1.tel 表示结构变量 student1 的电话信息。

和其他类型的变量一样，结构变量也可以进行初始化。结构初始化的一般形式如下：

```
struct 结构类型名 结构变量 =
{ 初始化数据 1, …… 初始化数据 n };
```

例如对变量 student1 的初始化可以用如下形式：

```
struct address student1 =
{ "wang","Road 1", 123456, 1111 };
```

2.10.2 结构体数组

结构体数组是一个数组，其数组的每一个元素都是结构体类型。在实际应用中，经常用结构体数组来表示具有相同数据结构的一个群体，如一个班的学生档案，一个车间职工的工资表等。定义结构体数组和结构体变量相仿，只需说明它为数组类型即可。

比如定义一个结构体数组 student，包含 3 个元素：student[0]、student[1]、student[2]，每个数组元素都具有 struct address 的结构形式，并对该结构体数组进行初始化赋值：

```
struct address
{
char name[30];      /*姓名，字符数组作为结构体中的成员 */
char street[40];    /*街道*/
```

```
unsigned long tel;      /*电话，无符号长整型作为结构体中的成员 */
unsigned long zip;     /*邮政编码*/
}student[3]={
{"Zhang","Road NO.1",111111,4444},
{"Wang"," Road NO.2",222222,5555},
{"Li"," Road NO.3",333333,6666}
}
```

> **说 明**
>
> 当对全部元素进行初始化赋值时，也可不给出数组的长度。

访问结构体数组成员的一般格式为：

```
结构数组名[下标].成员名
```

比如通过语句：

```
printf("%s",student[1].name);
```

便可实现打印 student 数组中第 1 个元素的 name 成员值。

2.10.3 指向结构体的指针

当一个指针用来指向一个结构体变量时，称之为结构体指针变量。结构体指针变量中的值是所指向的结构变量的首地址，通过结构指针即可访问该结构变量。这与数组指针和函数指针的情况是相同的。结构体指针变量定义的一般形式为：

```
struct 结构类型名 *结构指针变量名
```

例如，我们在 2.12.1 节中定义了 struct address 结构类型，如要定义一个指向该结构类型的指针变量 pstu，可写为：

```
struct address *pstu;
```

当然也可在定义 struct address 结构类型时同时说明 pstu。与前面讨论的各类指针变量相同，结构指针变量也必须要先赋值后使用。

赋值是把结构变量的首地址赋予该指针变量，不能把结构名赋予该指针变量。如果 student1 是被说明为 struct address 类型的结构变量，则：

```
pstu = &student1;
```

就是对结构指针进行赋值。有了结构指针变量，就能更方便地访问结构变量的各个成员。其访问的一般形式为：

```
 (*结构指针变量).成员名
```

或者：

```
结构指针变量->成员名
```

例如：

```
(*pstu).name
```

或者：

```
pstu->name
```

都是对 student1 结构体 name 成员的访问。

注 意

(*pstu)两侧的括号不可少，因为成员符“.”的优先级高于“*”，如去掉括号写作*pstu.num 则等效于*(pstu.num)，这样，意义就完全不对了。

通过总结，不难发现，我们可以使用以下 3 种方式访问结构体中的成员：一是结构变量.成员名；二是(*结构指针变量).成员名；三是结构指针变量->成员名。这 3 种用于表示结构成员的形式是完全等效的。

2.10.4 共用体

在 C 语言中，允许几种不同类型的变量存放到同一段内存单元中，也就是使用覆盖技术，几个变量互相覆盖。这种几个不同的变量共同占用一段内存的结构，被称为共用体类型结构，简称共用体。一般定义形式为：

```
union 共用体名
{
    数据类型   成员名 1;
    数据类型   成员名 2;
    ……
    数据类型   成员名 n;
}变量名表列;
```

只有先定义了共用体变量，才能在后续的程序中引用它。不能直接引用共用体变量，而只能引用共用体变量中的成员。引用方法如下：

```
共用体变量名.成员名
```

共用体类型数据具有以下特点：

- 同一个内存段可以用来存放几种不同类型的成员，但是在每一瞬间只能存放其中的一种，而不是同时存放几种。换句话说，每一瞬间只有一个成员起作用，其他的成员不起作用，即不是同时都存在和起作用的。
- 共用体变量中起作用的成员是最后一次存放的成员，在存入一个新成员后，原有成员就失去作用。
- 共用体变量的地址和它各成员的地址都是同一地址。
- 不能对共用体变量名赋值，也不能企图引用变量名来得到一个值，并且，不能在定义共用体变量时对它进行初始化。

➢ 不能把共用体变量作为函数参数，也不能是函数返回共用体变量，但可以使用指向共用体变量的指针。
➢ 共用体类型可以出现在结构体类型的定义中，也可以定义共用体数组。反之，结构体也可以出现在共用体类型的定义中，数组也可以作为共用体的成员。

程序 2.25 是一个关于共用体的定义与使用的例子。程序中首先声明一个名为 date 的共用体，并定义了共用体变量 a，然后根据输入的字符判断对共用体的哪一个成员变量进行赋值，最后输出该值。

【程序 2.25】共用体的定义与使用：test25.c。

```
#include <stdio.h>
main()
{
        char i;
        union date                              /*声明共用体数据类型*/
        {
            int day;                            /*共用体中的成员变量*/
            char month[20];
        int year;
        }a;                                     /*定义共用体变量 a*/
        scanf("%c",&i);                         /*输入判断字符 i*/
        if(i=='d') scanf("%d",&a.day);          /*若为 d，则输入的是 day 成员的值*/
        else if(i=='m') scanf("%s",&a.month);   /*若为 m，则输入的是 month 成员的值*/
        else if(i=='y') scanf("%d",&a.year);    /*若为 y，则输入的是 year 成员的值*/
        else printf("error input!\n");          /*错误字符*/
        if(i=='d') printf("a.day=%d\n",a.day);  /*下面是输出共用体变量 a 的某个成员*/
        if(i=='m') printf("a.month=%s\n",a.month);
        if(i=='y') printf("a.year=%d\n",a.year);
}
```

程序运行结果如下(↙表示回车)：

```
d↙
30↙
a.day=30
m↙
September↙
a.month= September
y↙
2009↙
a.year=2009
```

需要提醒读者的是，由于同一时刻只能使用共用体变量中的某一个成员变量，而不是同时使用几个，所以程序 2.25 中必须使用判断字符 i，以判断将要使用共用体中的哪一个成员(day、month 或者是 year)。

2.10.5 使用 typedef 定义类型

在 C 语言中，除系统定义的标准类型和用户自定义的结构体、共用体等类型之外，还可以

使用类型说明语句 typedef 定义新的类型来代替已有的类型。typedef 语句的一般形式是：

```
typedef 已定义的类型 新的类型;
```

例如：

```
typedef int INTEGER;         /*指定用 INTEGER 代表 int 类型*/
typedef float REAL;          /*指定用 REAL 代表 float 类型*/
```

在具有上述 typedef 语句的程序中，下列语句就是等价的：

```
int i, j; 与 INTEGER i, j;
float pi; 与 REAL pi;
```

2.11 链表

现实生活中存在大量需要动态存储和表示的数据，例如排队、数据排序等，这些问题都需要用链表的方式表示和处理。将对链表进行详细的介绍。

2.11.1 链表概述

链表是一种动态的数据结构。它是动态进行存储分配的一种结构。通常，对于大批的数据可以采用数组的方式保存，但使用数组保存存在明显的问题。首先，在 C 语言中，数组的大小在使用之前必须是确定的，一旦数据增加超过了数组的容量，就会发生数组溢出。为了保证不会发生数组溢出，当使用之前不能确定数组规模时，往往需要开设一个很大的数组，从而造成空间的浪费。如果在程序中采用动态数组的方式，即在数组增长的时候重新分配内存，然后将原始数据复制到新的数组中，这虽然是一种可行的办法，但效率太低。第二，如果要在数组中删除数据，就要将数组中删除点之后的数据向前移动；如果要在数组中插入数据，则必须将插入点后的元素向后移动。这种数据移动方式的效率同样是比较低的。链表正是针对数组的这些缺点而设计的一种存储数据的动态数据结构。

在链表中，所有数据元素都分别保存在一个具有相同数据结构的节点中，节点是链表的基本存储单位，一个节点与一个数据元素对应。每个节点在内存中使用一块连续的存储空间，每个节点可以使用不连续的存储空间，节点之间通过指针连在一起，连接节点的指针也称为链。

节点的存储结构在内存空间中通常分为两个部分：信息数据部分(也称为数据域)和连接节点的指针(也称为指针域)。节点定义采用结构体类型，一般形式为：

```
struct node
{
        datatype    data;          /*信息数据，根据实际数据定义*/
struct node * link;                /*指向节点 node 的指针*/
};
```

在这里，节点的数据类型名称是 struct node，data 是实际需要的结构成员分量，datatype

是实际分量所需要的数据类型，link 是一个指向 struct node 类型的结构指针，即 link 指向的对象是一个同样类型的数据节点，它是一个动态的指针，用来存放下一个节点的地址，通过 link 指针，一个个节点被依次连接起来，形成链表。

一个链表一般由头指针、表头节点和数据节点三部分组成。它们之间的关系如图 2.11 所示。

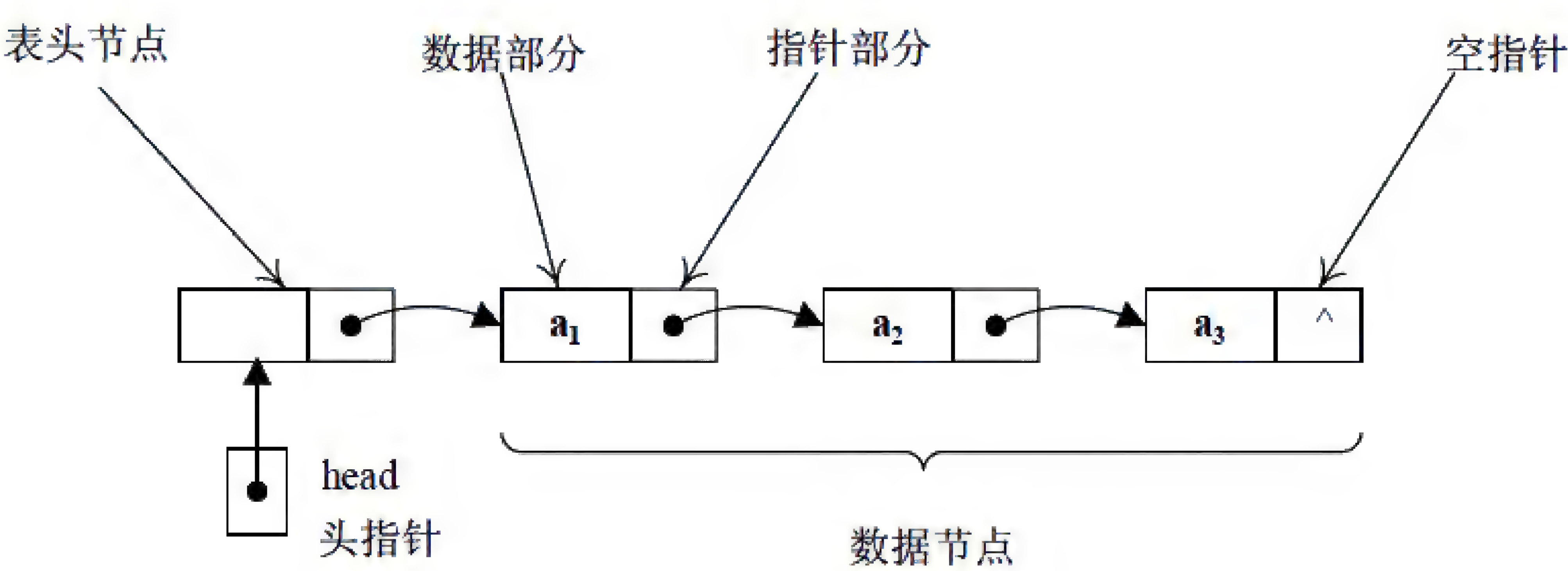


图 2.11　单链表结构

2.11.2　建立动态单向链表

建立链表首先要定义一个包含数据域和指针域的结构类型，然后建立指向表头节点的头指针 head，最后通过 malloc 函数动态申请一块内存作为表头节点。

```
typedef struct node
{
        int data;               /*信息*/
        struct node *link;      /*指针*/
}NODE;                          /*定义节点*/
NODE *head;                     /*定义头指针 head */
```

定义结构类型和头节点之后，我们要建立不包含数据的表头节点，可以按下列语句进行操作。

```
NODE *p;                              /*说明一个指向节点的指针变量 p */
p = (NODE*) malloc(sizeof(NODE));     /*申请表头节点*/
p->link = NULL;                       /*将表头节点的 link 置为 NULL */
head = p;                             /*head 指向表头节点 p*/
```

此时链表的状态如图 2.12 所示，由于此时链表中只有一个表头节点，没有数据节点，所以称为空链表。

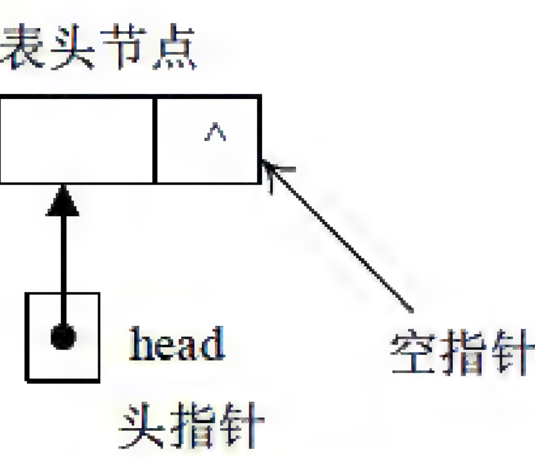


图 2.12　空链表

为了在链表中保存数据，可以从表头位置将数据节点插入到链表中，例如，插入一个数据节点：

```
p = (NODE*) malloc(sizeof(NODE));        /*申请一个数据节点*/
gets(p ->data);                          /*输入一个新的数据*/
p->link = head->link;                    /*建立链接关系。将表头节点的 link 存入 p 的 link 中*/
head->link = p;                          /*将数据节点插在表头节点之后成为第一个数据节点*/
```

插入第一个数据节点后链表如图 2.13 所示，然后继续插入下一个数据节点。

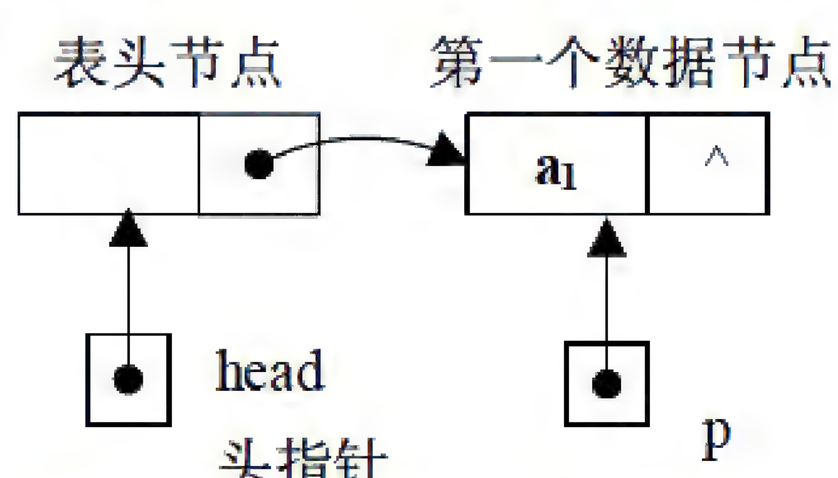


图 2.13　插入一个节点后的链表

根据上面的链表建立过程，可以写出函数 create 建立有 n 个数据节点的链表，如下所示：

```
create(NODE *head,int n)
{
        NODE *p;
for(; n>0;n--)
{
            p = (NODE*) malloc(sizeof(NODE));
            if(p==NULL)
                exit(0);
            gets(p->data);
            p->link = head->link;
            head->link = p;
}
}
```

2.11.3　单向链表的输出

将单向链表中各节点依次输出，首先要知道链表第一个节点的地址，然后设一个指针变量 p，先指向第一个节点，输出 p 所指的节点，然后使 p 后移一个节点再输出。直到链表的尾节点。如下面这段代码：

```
void print(NODE *head)
{
NODE *p;
        p = head;
        if(head!= NULL)
        do
        {
printf("%d /n", p->data);
```

```
            p = p->link;
        }while(p!=NULL)
}
```

2.11.4 对单向链表的删除操作

要删除链表中第 i 个节点的基本算法如下：

(1) 定位第 i-1 个节点。指针 q 指向第 i-1 个节点，指针 p 指向被删除的节点。

(2) 摘链。q->link = p->link。

(3) 释放 p 节点。free(p)。

具体操作过程如图 2.14 所示。如图 2.15 所示显示了删除第 i 个节点后链表的状态。

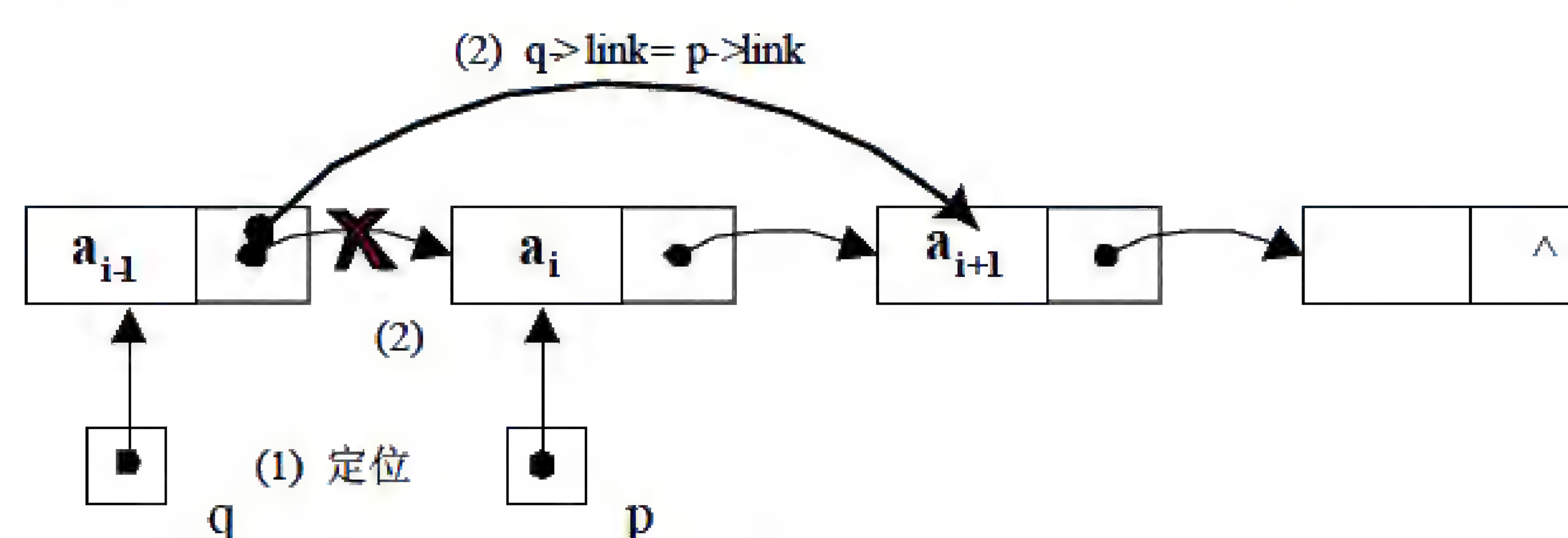


图 2.14　删除第 i 个节点的过程

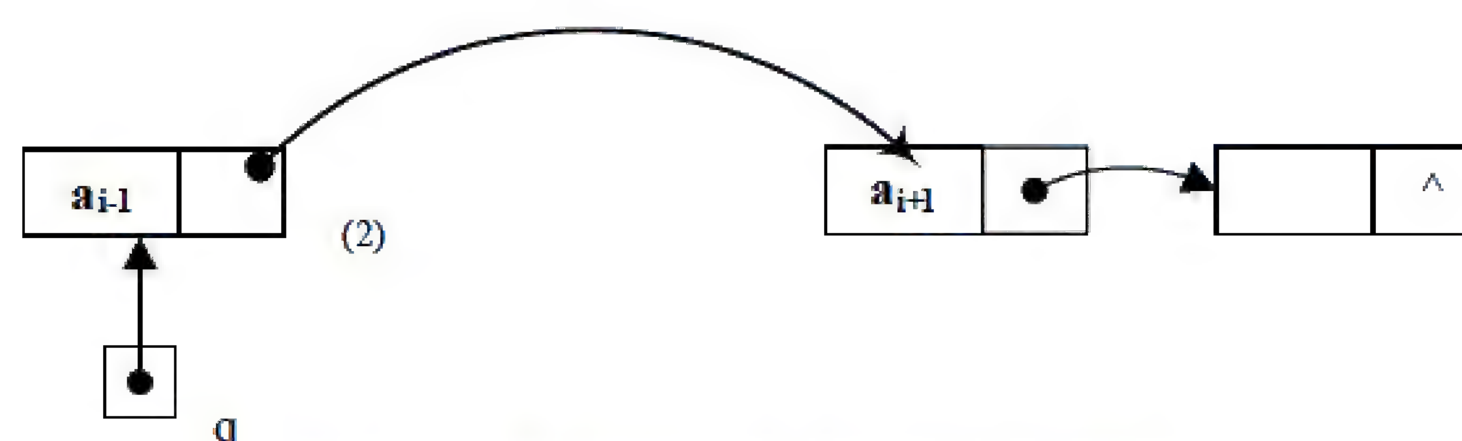


图 2.15　删除第 i 个节点后链表的状态

根据上述算法的基本思想，可以写出删除链表中第 i 个节点的程序如下：

```
delete_node(NODE *head, int i)
{
NODE *q, *p;
int n;
for(n=0,q=head;n<i-1&&q->link!=NULL;++n)
        q = q->link;                    /*(1)定位第 i-1 个节点*/
if(i<0&&q->link!=NULL)
        {
p = q->link;                /*p 指向被删除的第 i 个节点*/
            q->link = p->link;          /*(2)摘链*/
            free(p);                    /*(3)释放 p 节点*/
        }
}
```

2.11.5　对单向链表的插入操作

在链表的第 i 个节点的后面插入一个新节点的基本算法如下。

(1) 定位第 i 个节点。让指针 q 指向第 i 个节点，指针 p 指向需要插入的节点。

(2) 链接后面指针。p->link = q->link。

(3) 链接前面指针。q->link=p。

具体操作过程如图 2.16 所示。如图 2.17 所示是在第 i 个节点之后插入新节点后链表的状态。

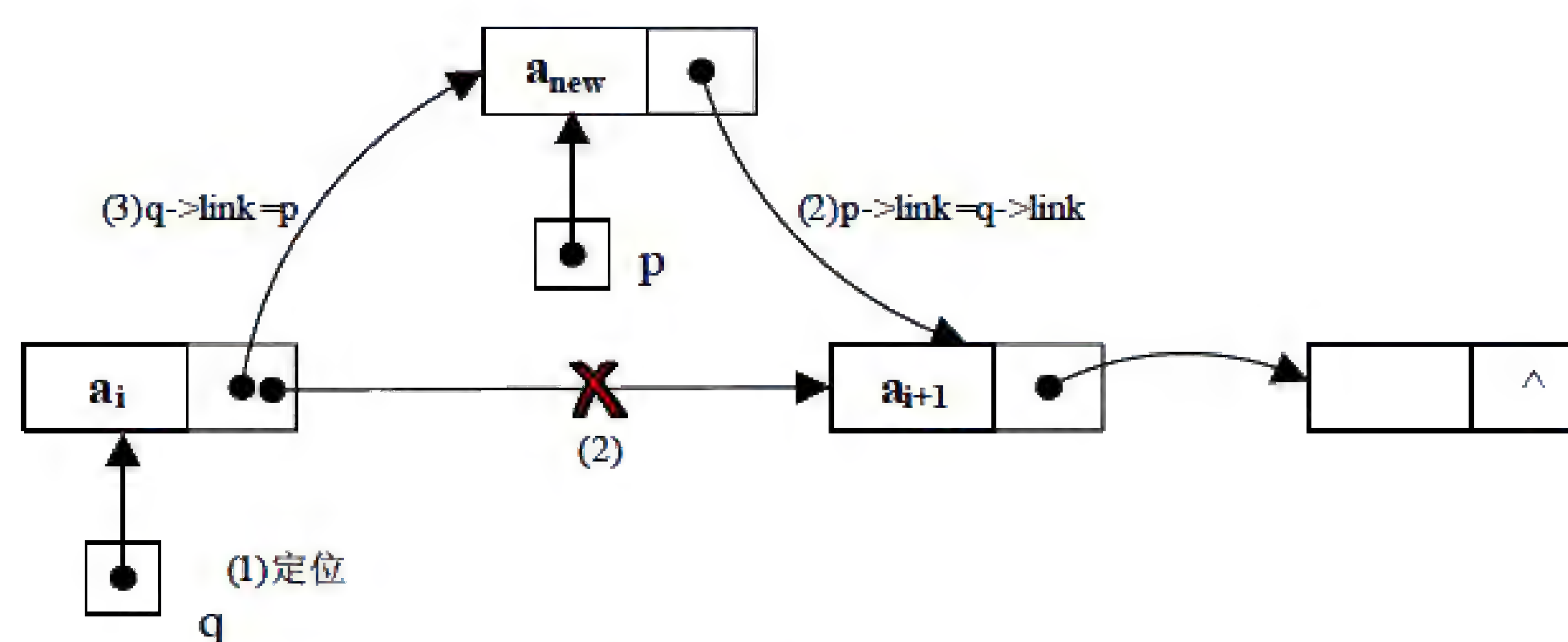


图 2.16　在第 i 个节点的后面插入新数据节点的过程

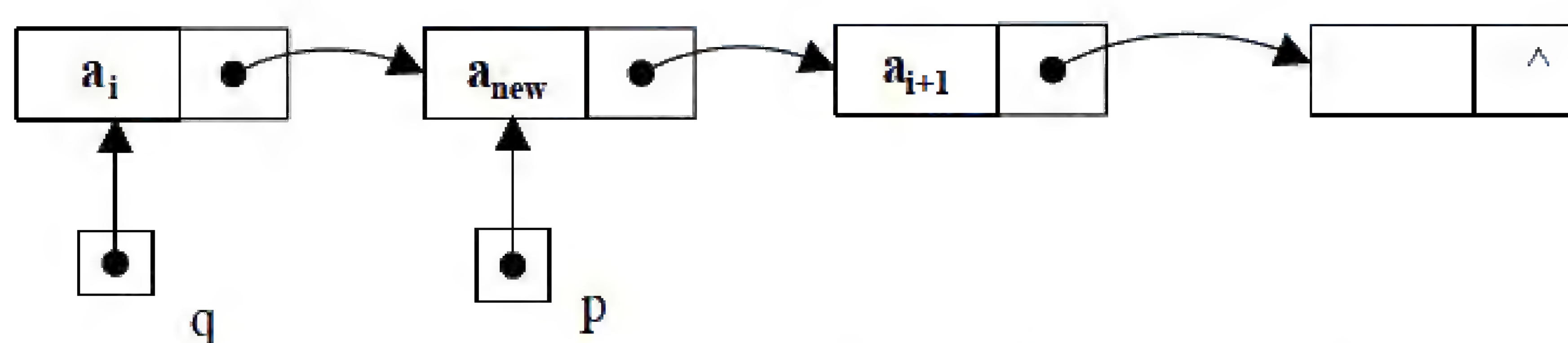


图 2.17　在第 i 个节点之后插入新数据节点后链表状态

根据上述算法的基本思想，可以写出在链表第 i 个节点之后插入一个新数据节点 p 的程序如下：

```
insert_node(NODE *head, NODE *p, int i)
{
NODE *q;
        int n=0;
for(q=head; n<i&&q->link!=NULL;++n)
          q=q-link;               /*(1)定位第 i 个节点*/
p->link=q->link;                  /*(2)链接后面指针*/
q->link=p;                        /*(3)链接前面指针*/
}
```

2.11.6　循环链表

循环链表是另一种形式的表示线性聚集的链表，它的节点与单链表相同，与单链表不同的是链表中表尾节点的指针域中不是 NULL，而是存放了一个指向链表表头节点的指针，这样，只要知道表中任何一个节点的地址，就能遍历表中其他任一节点，如图 2.18 所示。

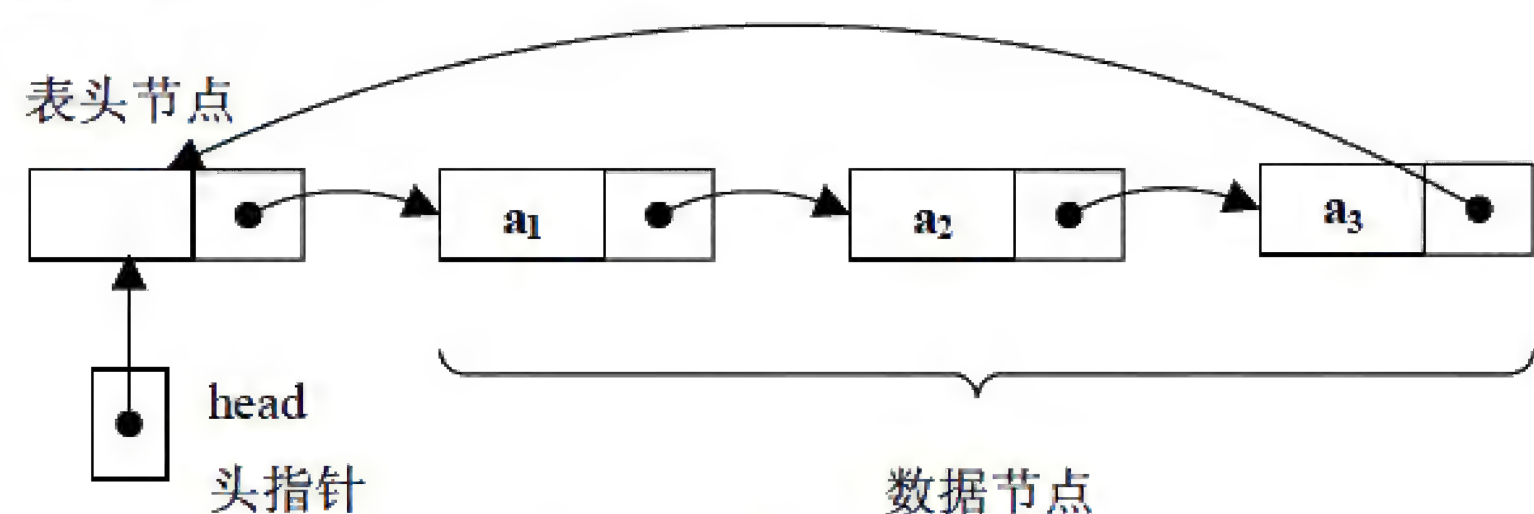


图 2.18　循环链表

循环链表的运算与单链表类似，但在涉及链头与链尾处理时稍有不同。例如，在实现循环链表的插入运算时，如果是在表的最前端插入，必须改变链尾最后一个节点的 link 域的值，这就需要搜索到最后一个节点。

2.11.7　双向链表

在单链表中，搜索一个指定节点的后继节点非常方便，只要该节点的 link 域的内容不为空，就可以通过 link 域找到该节点的后继节点地址。但是要搜索一个指定节点的前驱节点十分不容易，必须从链头开始，沿着 link 链顺序检测，直到某一节点的后继节点为该指定节点，则此节点即为该节点的前驱节点。为克服这一缺点，可以考虑双向链表。

在双向链表的每个节点中，应有两个链接指针作为它的数据成员：lLink 指示它的前驱节点，rLink 指示它的后继节点。因此双向链表的每个节点至少有 3 个域：

lLink(左链指针)	Data(数据)	rLink(右链指针)

节点之间的链接关系如图 2.19 所示。

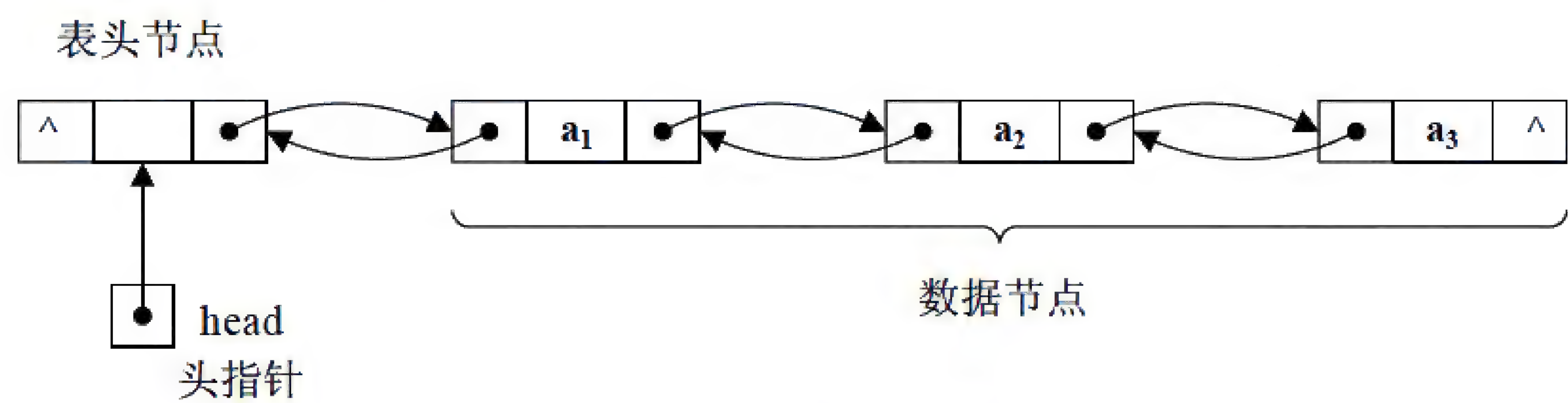


图 2.19　带表头的双向链表

指针 p 指向双向循环链表的某一节点，那么，p->lLink 指示 p 所指节点的前驱节点，p->lLink->rLink 中存放的是 p 所指前驱节点的后继节点的地址，即 p 所指节点本身。同样的，p->rLink 指示 p 所指节点的后继节点，p->rLink->lLink 也指向 p 节点本身。因此有 p= = p->lLink->rLink == p->rLink->lLink，其过程如图 2.20 所示。

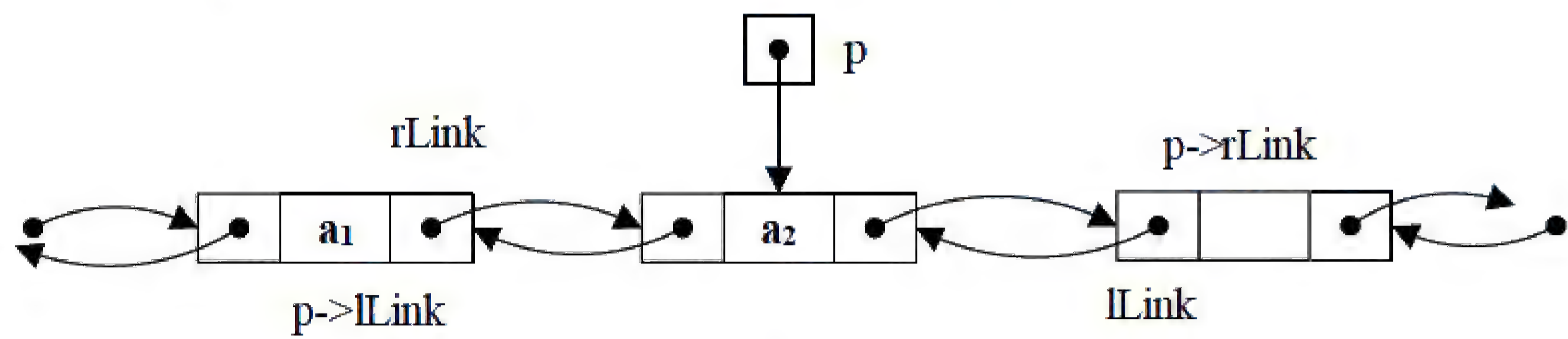


图 2.20　节点指针的指向

双向链表的插入、删除操作的思路与单向链表是一样的，但操作时稍复杂一些。

如果要在当前指针 current 所指节点后面插入一个新节点 p，需修改 5 个指针，把新节点链入两个方向的链中，具体操作如下：

```
p->lLink = current;   p->rLink = current->rLink;   /*改新节点的两个链域*/
current->rLink = p;   current = current->rLink;    /*改前驱节点的后继链域*/
current->rLink->lLink = current;                   /*改后继节点的前驱链域*/
```

如果要删除双向链表中的一个节点，删除过程分为如下两步：

第一步先把当前节点从链中分离出来，修改前驱节点的后继指针和后继节点的前驱指针。

```
current->rLink->lLink = current->lLink;      /*从 lLink 链中摘下*/
current->lLink->rLink = current->rLink;      /*从 rLink 链中摘下*/
```

第二步再把当前节点释放。

2.12 位运算符和位运算

位运算是一种 C 语言提供的对二进制位的操作功能。它应用于整型数据，即把整型数据看成固定的二进制序列，然后对这些二进制序列进行按位运算。C 语言提供了 6 种基本位运算功能：按位与、按位或、取反、异或、左移、右移，下面将对其进行介绍。

2.12.1 “按位与”运算符(&)

按位与运算是指对两个运算量相应的位进行逻辑与，“&”的运算规则与逻辑与“&&”相同。按位与表达式为 c = a & b，运算规则如图 2.21 所示。

	a:	1010,	1001,	0101,	0111
&	b:	0110,	0000,	1111,	1011
	c:	0010,	0000,	0101,	0011

图 2.21　按位与运算

2.12.2 “按位或”运算符(|)

按位或运算是指对两个运算量相应的位进行逻辑或，“|”的运算规则或逻辑与“||”相同。按位或表达式为 c = a | b，运算规则如图 2.22 所示。

	a:	1010,	1001,	0101,	0111
|	b:	0110,	0000,	1111,	1011
	c:	1110,	1001,	1111,	1111

图 2.22　按位或运算

2.12.3 “取反”运算符(~)

按位取反运算是将二进制表示的运算对象按位取反，即将 1 变为 0，将 0 变为 1。按位取反表达式为 c=~a，运算规则如图 2.23 所示。

~	a:	1010,	1001,	0101,	0111
	c:	0101,	0110,	1010,	1000

图 2.23　取反运算

2.12.4 “异或”运算符(^)

按位异或运算的规则是：两个运算量的相应位相同，则结果为 0，相异则结果为 1。按位异或表达式为 c = a ^ b，运算规则如图 2.24 所示。

	a:	1010,	1001,	0101,	0111
^	b:	0110,	0000,	1111,	1011
	c:	1100,	1001,	1010,	1100

图 2.24　按位异或运算

2.12.5 移位运算符(<<和>>)

左移和右移是把整数作为二进制位序列，求出把这个序列左移若干位或者右移若干位后得到的序列。它们的一般形式为：x<<n 或 x>>n(x 是要被移位的量；n 是要移动的位数)。

左移运算规则是将 x 的二进制位全部向左移动 n 位，将左边移出的高位舍弃，右边空出的低位补 0。右移运算是将 x 的二进制位全部向右移动 n 位，将右边移出的低位舍弃，左边高位空出要根据原来量符号位的情况进行补充。对无符号数则补 0；对有符号数，若为正数则补 0，若为负数则补 1。

例如，设 a=5，则：

(1) b = a<<3 即 b = 0000,0101<<3 = 0010,1000 = 40。

(2) c = a>>2 即 c = 0000,0101>>2 = 0000,0001 = 1。

另外，左移运算等效于将整数值乘以 2 的幂；右移运算等效于将整数值除以 2 的幂，幂的大小即为左移或右移的位数。

2.12.6 位域

有些信息在存储时，并不需要占用一个完整的字节，而只需占几个或一个二进制位。例如在存放一个开关量时，只有 0 和 1 两种状态，用 1 位即可。为了节省存储空间，并使处理简便，C 语言又提供了一种数据结构，称为“位域”。

所谓“位域”，就是把一个字节中的二进位划分为几个不同的区域，并说明每个区域的位数，每个域有一个域名，允许在程序中按域名进行操作。这样就可以把几个不同的对象用一个字节的二进制位域来表示。位域的定义和位域变量的定义形式为：

```
struct 位域结构名
{
类型说明符 位域名 1: 位域长度;
```

```
类型说明符 位域名 2: 位域长度;
……
类型说明符 位域名 n: 位域长度;
};
```

为了节省内存空间，可以把几个数据压缩到少数的几个类型空间中，比如需要表示两个 3 位的二进制数和一个 2 位的二进制数，则可以用一个 8 位的字符表示。如下所示定义一个 8 位的位域：

```
struct
{
char a : 3;
char b : 3;
char c : 2;
};
```

可以看到，这个结构体所占空间为一个字节(8 位)，节省了内存空间。

位域的说明与结构变量说明的方式相同，可采用先定义后说明、同时定义说明或者直接说明这 3 种方式。使用位域时应注意以下几点：

- 一个位域必须存储在同一个字节中，不能跨两个字节。
- 位域的长度不能大于一个字节的长度，也就是说不能超过 8 位。
- 可以定义无名位域，这时它只能用来做填充或调整位置，无名位域在程序中是不能使用的。

2.13 C 语言预处理命令

预处理命令可以改变程序设计环境，提高编程效率，它们并不是 C 语言本身的组成部分，不能直接对它们进行编译，必须在对程序进行编译之前，先对程序中这些特殊的命令进行“预处理”。经过预处理后，程序就不再包括预处理命令了，最后再由编译程序对预处理之后的源程序进行编译处理，得到可供执行的目标代码。C 语言提供的预处理功能有 3 种，分别为宏定义、文件包含和条件编译，下面将对它们进行简单介绍。

2.13.1 宏定义

在 C 语言源程序中允许用一个标识符来表示一个字符串，称为“宏”，被定义为“宏”的标识符称为“宏名”。在编译预处理时，对程序中所有出现的宏名，都用宏定义中的字符串去代换，这称为“宏代换”或“宏展开”。

宏定义是由源程序中的宏定义命令完成的，宏代换是由预处理程序自动完成的。在 C 语言中，宏分为有参数和无参数两种。无参宏的宏名后不带参数，其定义的一般形式为：

```
#define 标识符 字符串;
```

其中的“#”表示这是一条预处理命令(在 C 语言中凡是以“#”开头的均为预处理命令)，

“define”为宏定义命令，“标识符”为所定义的宏名，“字符串”可以是常数、表达式、格式串等。符号常量的定义就是一种无参宏定义。

此外，常对程序中反复使用的表达式进行宏定义。例如：

```
#define M (y*y+3*y);
```

它的作用是指定标识符 M 来代替表达式(y*y+3*y)。在编写源程序时，所有的(y*y+3*y)都可由 M 代替，而对源程序进行编译时，将先由预处理程序进行宏代换，即用(y*y+3*y)表达式去置换所有的宏名 M，然后再进行编译。

C 语言允许宏带有参数。在宏定义中的参数称为形式参数，在宏调用中的参数称为实际参数。对于带参数的宏，在调用中，不仅要宏展开，而且要用实参去代换形参。

带参宏定义的一般形式为：

```
#define  宏名(形参表)  字符串;
```

在字符串中含有各个形参。

带参宏调用的一般形式为：

```
宏名(实参表);
```

例如：

```
#define M(y) y*y+3*y          /*宏定义*/
……
k=M(5);                       /*宏调用*/
……
```

在上面的宏调用时，用实参 5 去代替形参 y，经预处理宏展开后的语句为：

```
k=5*5+3*5;
```

程序 2.26 给出了一个宏定义和调用的完整实例。

【程序 2.26】定义一个名为 MAX 的带参数的宏，可以通过用它来选出参数 a、b 中的较大值：test26.c。

```
#include <stdio.h>
#define MAX(a,b) (a>b)?a:b        /*带参数的宏定义*/
main()
{
    int x,y,max;
    printf("input two numbers: ");
    scanf("%d %d",&x,&y);
    max=MAX(x,y);                 /*宏调用*/
    printf("max=%d\n",max);
}
```

程序运行结果如下(□表示空格，↙表示回车)：

```
input two numbers: 2009□2010↙
max=2010
```

可以看到，宏替换相当于实现了一个函数调用的功能，而事实上，与函数调用相比，宏调用更能提高 C 程序的执行效率。

2.13.2 文件包含

文件包含是 C 预处理程序的另一个重要功能，文件包含命令行的一般形式为：

```
#include "文件名"
```

或者

```
#include <文件名>
```

文件包含命令的功能是把指定的文件插入该命令行位置取代该命令行，从而把指定的文件和当前的源程序文件连成一个源文件。

在程序设计中，文件包含是很有用的。一个大的程序可以分为多个模块，由多个程序员分别编程，有些公用的符号常量或宏定义等可单独组成一个文件，在其他文件的开头用包含命令包含该文件即可使用。这样，可避免在每个文件开头都去书写那些公用量，从而节省时间，并减少出错。

这里对 C 语言的文件包含命令进行以下几点说明：

(1) 包含命令中的文件名可以用双引号引起来，也可以用尖括号括起来。例如以下写法都是允许的：

```
#include "stdio.h"
#include <stdio.h>
```

但是这两种形式是有区别的：使用尖括号表示在包含文件目录中去查找(包含目录是由系统的环境变量进行设置的，一般为系统头文件的默认存放目录，比如 Linux 系统在/usr/include 目录下)，而不在源文件的存放目录中查找；使用双引号则表示首先在当前的源文件目录中查找，若未找到才到包含目录中去查找。用户编程时可根据自己文件所在的目录来选择某一种命令形式。

(2) 一个 include 命令只能指定一个被包含文件，若有多个文件要包含，则需用多个 include 命令。

(3) 文件包含允许嵌套，即在一个被包含的文件中又可以包含另一个文件。

2.13.3 条件编译

预处理程序提供了条件编译的功能，可以按不同的条件去编译不同的程序部分，因而产生不同的目标代码文件，这对于程序的移植和调试是很有用的。条件编译可分为 3 种形式。第一种形式如下：

```
#ifdef 标识符
  程序段 1
#else
  程序段 2
#endif
```

它的功能是如果标识符已被 #define 命令定义过则对程序段 1 进行编译；否则对程序段 2 进行编译。如果没有程序段 2(为空)，本格式中的#else 可以没有，即可以写为：

```
#ifdef 标识符
        程序段
#endif
```

第二种形式如下：

```
#ifndef 标识符
        程序段 1
#else
        程序段 2
#endif
```

与第一种形式的区别是将“ifdef”改为“ifndef”。它的功能是如果标识符未被#define 命令定义过则对程序段 1 进行编译，否则对程序段 2 进行编译。这与第一种形式的功能正好相反。

第三种形式如下：

```
#if 常量表达式
     程序段 1
#else
    程序段 2
#endif
```

它的功能是如果常量表达式的值为真(非 0)，则对程序段 1 进行编译，否则对程序段 2 进行编译。因此可以使程序在不同的条件下完成不同的功能。

2.13.4 #error 等其他常用预处理命令

除了上面介绍的之外，C 语言还有#erroe、#line、#pragma 等其他常用的预处理命令，在很多 C 语言的程序中也是经常可见的。下面向读者简单介绍一下它们。

1. #error

#error 指令强制编译程序停止编译，它主要用于程序调试。#error 指令的一般形式是：

```
#error error-message
```

注意，宏串 error-message 不用双引号引起来。遇到#error 指令时，错误信息被显示，可能同时还显示编译程序作者预先定义的其他内容。

2. #line

#line 指令改变__LINE__和__FILE__的内容。__LINE__和__FILE__都是编译程序中预定义的标识符。标识符__LINE__的内容是当前被编译代码行的行号，__FILE__的内容是当前被编译源文件的文件名。#line 的一般形式是：

```
#line number "filename"
```

其中，number 是正整数并变成__LINE__的新值；可选的“filename”是合法文件标识符并

变成__FILE__的新值。#line 主要用于调试和特殊应用。

3. #pragma

#pragma 是编译程序实现时定义的指令，它允许由此向编译程序传入各种指令。例如，一个编译程序可能具有支持跟踪程序执行的选项，此时可以用#pragma 语句选择该功能，编译程序忽略其不支持的#pragma 选项。使用#pragma 预处理命令可提高 C 源程序对编译程序的可移植性。

2.14 本章小结

本章较详细地讲解了 C 语言编程的基础知识，介绍的内容包括 C 语言的数据类型与运算规则、程序设计基本结构、数组、字符数据处理、指针、函数、结构体及其他构造类型、链表和预处理等。这些是在 Linux 下阅读和编写 C 程序的基础，读者务必熟练掌握和应用 C 语言。

实战演练

1. 编写一个程序，输出以下信息：

```
**********************************************************
Hello, Linux world!
**********************************************************
```

2. 编写一个程序，接受用户从键盘输入的字符，如果是小写字母则转换为大写字母，如果是大写字母，则原样输出。

3. 有一函数：

$$y=\begin{cases} x(x<1) \\ 2x-1(1\le x<10) \\ 3x-8(x\ge 10) \end{cases}$$

试编写一个 C 程序，输入 x，输出 y 值。

4. 编写一个程序，首先让用户在下面两个选项中选择一个：

A. 把温度从摄氏度转换为华氏度。

B. 把温度从华氏度转换为摄氏度。

然后提示用户输入温度值，输出转换后的新值。提示：把输入的值乘以 1.8，然后加 32，即可把摄氏度转换成华氏度。用输入的值减去 32，然后乘以 5，再用 9 除得到的结果，即可把华氏度转换成摄氏度。

5. 编写一个程序，从键盘读入 5 个 double 值，把它们存放到数组中。计算每个值的倒数(x 的倒数即 1.0/x)，然后把它们存储在另外一个数组中。输出这些倒数值，计算并输出倒数的和。

6. 有一个 3×4 的矩阵，编写一个 C 程序，求所有元素中的最大值，以及该元素所在的行号和列号。

7. 编写一个程序，使用递归函数计算用户输入整数的阶乘值。

8. 编写一个程序，计算从键盘输入的任意多个浮点数的平均数。把所有值存储在开始计算之前动态分配的内存中，然后显示平均数，注意不能要求用户先声明要输入多少个值。

9. 定义一个 struct 类型，存放一个人的名字和他的电话号码。在一个程序中使用这个 struct，该程序允许输入一个或多个人的名字和对应的电话号码，然后把它们存储在一个结构数组的元素中。该程序应该能接受输入第二个人名，并输出与这个名字对应的所有号码。

10. 设 ha 和 hb 分别是两个带表头节点的非递减有序单链表的表头指针，试设计一个算法，将这两个有序链表合并成一个非递减有序的单链表。要求结果链表仍使用原来两个链表的存储空间，不另外占用其他的存储空间。表中允许有重复的数据。

第3章

vim与Emacs编辑器

文本编辑器是计算机最基本的应用，修改配置文件、编写程序或者建立文件都需要用到它。Linux 提供了齐全的文本编辑器，可以让用户按照自己的喜好进行选择。在 Linux 众多的文本编辑器中，vim 和 Emacs 编辑器是 Linux 用户最为普遍的选择。本章将向读者介绍这两种文本编辑器。



本章内容：

- ◎ vim 编辑器的使用详解。
- ◎ vim 使用实例。
- ◎ Emacs 编辑器的使用详解。
- ◎ Emacs 使用实例。

3.1 vim 的使用

vim 简介

vim 是“Vim IMproved”的简称，其是 vi 编辑器的加强版，其提供了执行输入、输出、删除、查找、替换、块操作等众多文本操作，用户还可以根据自己的需要对其进行定制。

其是 UNIX/Linux 下最基本的文本编辑器，工作在字符模式下，由于不需要图形接口，使它成为效率很高的文本编辑器。尽管在 Linux 上也有很多图形接口的编辑器可用，但 vim 在系统和服务器管理应用中的功能是那些图形编辑器所无法比拟的，所以在本书也仅仅对 vim 的基础使用方法进行了较为详细的介绍，而对其他代码编辑器则仅仅进行概述。

1. vim 的启动和退出

在 Linux 终端命令提示符下输入 vim(或 vim+文件名)，即可启动 vim 编辑器。例如：

```
vim filename
```

或者

```
vim
```

按下“Enter”键后，Linux 便会自动打开文件名为“filename”文件的 vim 编辑接口，其启动接口如图 3.1 所示。



图 3.1　vim 的启动接口

当使用“vim+文件名”的命令来启动 vim 时，若进行编辑的是当前工作目录下已存在的文件，启动后即可看到该档中的内容；若是当前目录下不存在的文件，则系统首先创建该文件，再使用 vim 进行编辑。

要退出 vim，必须先按下“Esc”键回到 vim 的命令行工作模式(关于 vim 的工作模式请参考下一小节)，然后键入“:”，此时游标会停留在最下面一行(底行模式)，再键入“q”，最后按下“Enter”键即可退出 vim。

2. vim 工作模式及其切换

vim 拥有 3 种工作模式：命令行工作模式(command mode)、插入工作模式(input mode)与底行工作模式(last line mode)，这 3 种工作模式下的功能可描述如下：

- 命令行工作模式：也叫作“普通模式”，启动 vim 后默认进入此模式，在该模式下可以使用隐式命令(命令不显示)来实现游标的移动、复制、粘贴、删除等操作，但在该模式下，编辑器并不接受用户从键盘输入的任何字符来作为文件的编辑内容，也就是说并不能将 C 语言代码输入到文件。
- 插入工作模式：在该工作模式下，用户输入的任何字符都被认为是编辑到某一个档的内容，并直接显示在 vim 的文本编辑区，在该模式下可以将 C 语言代码输入到文件。
- 底行工作模式：在该工作模式下，用户输入的任何字符串都会被当作命令，会在 vim 的最下面一行显示，按下“Enter”键后便会执行该命令，如果该字符串并不是一个有效的命令，则会出现错误提示。

使用 vim 编辑器，首先必须能够熟练掌握各种工作模式的用途以及各种工作模式间的切换，图 3.2 所示为 vim 3 种工作模式间的切换方法。

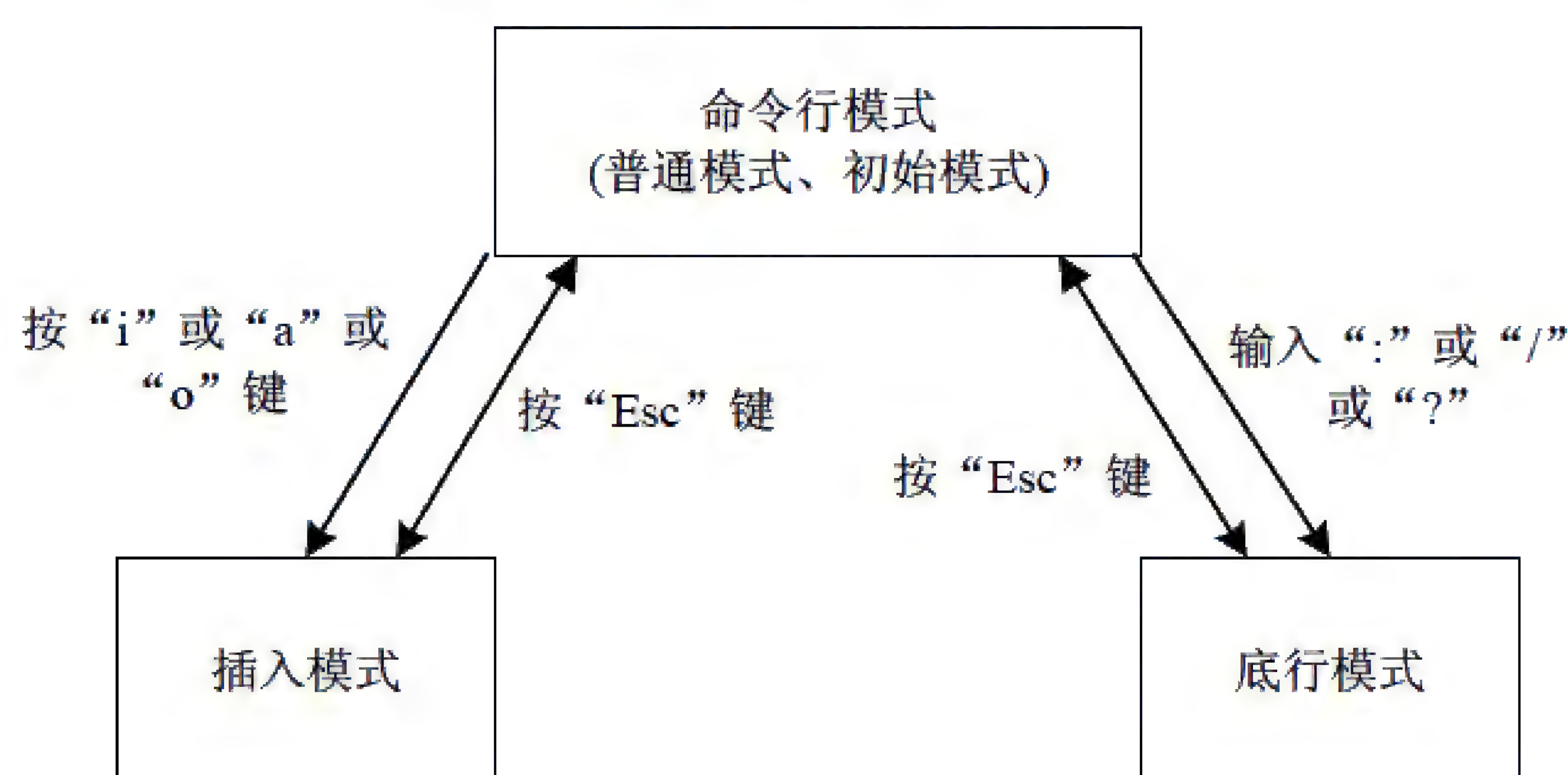


图 3.2　vim 3 种工作模式间的切换方法

从图 3.2 中可以看到，命令行工作模式是 vim 编辑器的初始模式，从该模式下可以实现到任何模式的切换；而插入模式和底行模式之间不能相互切换，因为在插入模式下，任何输入的字符都被认为是编辑到某一个档的内容，而不是命令；在底行模式下，任何输入的字符都被看作是底行命令(尽管可能是不合法的)，两者都必须先通过命令行模式才能进入对方，即需要先按下“Esc”键回到初始模式。

3. vim 的命令行工作模式

vim 在命令行工作模式下的主要操作是使用方向键或快捷键对当前游标进行定位以及使用相应的命令对当前档中的文本进行诸如复制、删除、粘贴等基础编辑操作，这些命令说明如表 3.1～表 3.4 所示。

注 意

命令行工作模式下的命令比较多，在此仅作简单介绍，用户在使用时也可以查阅帮助文档。

在命令行工作模式下，可以通过使用上、下、左、右 4 个方向键来移动游标的位置。但是在类似使用 telnet 远程登录等场合下就没法使用方向键，此时必须用命令行模式下的游标移动命令，这些命令对应的字符串和操作说明如表 3.1 所示。

表 3.1 移动游标的常用命令

命　令	操 作 说 明
h	向左移动游标
l	向右移动游标
j	向下移动游标
k	向上移动游标
^	将游标移动到该行的开头(指第一个非空字符上)
$	将游标移动到该行行尾，同键盘上的“End”键
0	将游标移动到该行行首，同键盘上的“Home”键
G	将光标移动到文件最后一行的开头(第一个非空字符)
nG	将光标移动到文件的第 n 行的开头(第一个非空字符)，n 为正整数
w	光标向后移动一个字(单词)
nw	光标向后移动 n 个字(单词)，n 为正整数
b	光标向前移动一个字(单词)
nb	光标向前移动 n 个字(单词)，n 为正整数
e	将游标移动到本单词的最后一个字符。如果游标所在的位置为本单词的最后一个字符，则跳动到下一个单词的最后一个字符。“.”、“,”、“#”、“/”等特殊字符都会被当成一个字
{	游标移动到前面的“{”处。这在使用 vim 进行 C 语言编程时很适用
}	同“{”的使用，将游标移动到后面的“}”处
Ctrl+b	向上翻一页，相当于 Page Up
Ctrl+f	向下翻一页，相当于 Page Down
Ctrl+u	向上移动半页
Ctrl+d	向下移动半页
Ctrl+e	向下翻一行
Ctrl+y	向上翻一行

复制、粘贴是在编辑文档时最常用的操作之一，可以大大节约用户重复输入的时间。vim 的命令行工作模式下常用的复制、粘贴命令对应的字符串和操作说明如表 3.2 所示。

表 3.2 复制粘贴的常用命令

命　令	操 作 说 明
yy	复制游标所在行的整行内容
yw	复制游标所在的单词的内容
nyy	复制从光标所在行开始向下的 n 行内容，n 为正整数，表示复制的行数

(续表)

命　令	操 作 说 明
nyw	复制从游标所在字开始向后的 n 个字，n 为正整数，表示复制的字数
p	粘贴，将复制的内容粘贴在游标所在的位置

在 vim 编辑器中，可以一次删除一个字符，也可以一次删除多个字符和整行，vim 命令行工作模式下常用的删除命令对应的字符串和操作说明如表 3.3 所示。

表 3.3　删除文本的常用命令

命　令	操 作 说 明
x	删除游标所在位置的字符，同键盘上的“Delete”键
X	删除游标所在位置的前一个字符
nx	删除游标所在位置及其后的 n-1 个字符，n 为正整数
nX	删除游标所在位置及其前的 n-1 个字符，n 为正整数
dw	删除游标所在位置的单词
ndw	删除游标所在位置及其后的 n-1 个单词，n 为正整数
d0	删除当前行游标所在位置前的所有字符
d$	删除当前行游标所在位置后的所有字符
dd	删除游标所在行
ndd	删除游标所在行及其向下的 n-1 行，n 为正整数
nd+上方向键	删除游标所在行及其向上的 n 行，n 为正整数
nd+下方向键	删除游标所在行及其向下的 n 行，n 为正整数

vim 在命令行工作模式还提供了一些其他常用的命令，包括字符替换、撤销操作、符号匹配等，其对应的字符串和操作说明如表 3.4 所示。

表 3.4　其他常用命令

命　令	操 作 说 明
r	替换游标所在位置的字符，例如 rx 是指将游标所在位置的字符替换为 x
R	替换游标所到之处的字符，直到按下“Esc”键为止
u	表示复原功能，即撤销上一次操作
U	取消对当前行所做的所有改变
.	重复执行上一次的命令
ZZ	保存文档后退出 vim 编辑器
%	符号匹配功能，在编辑时若输入“%(”，系统会自动匹配相应的“)”

4. vim 的插入工作模式

在插入工作模式下 vim 没有繁琐的命令，用户从键盘输入的任何有效字符都被看作是写进当前正在编辑的档中的内容，并显示在 vim 的文本编辑区，也就是说，只有在插入模式下，才可以进行文字的输入操作。表 3.5 所示为从命令行模式切换至插入模式的几个常用命令，当插入工作模式下时，随时可以使用“Esc”键回到 vim 的命令行工作模式。

表 3.5　命令行工作模式切换至插入工作模式的命令

命　令	操 作 说 明
i	从游标所在的位置开始插入新的字符
I	从光标所在行的行首开始插入新的字符
a	从游标所在位置的下一个字符开始插入新的输入字符
A	从光标所在行的行尾开始插入新的字符
o	新增加一行，并将游标移动到下一行的开头开始插入字符
O	在当前行的上面新增加一行，并将游标移动到上一行的开头开始插入字符

5. vim 的底行工作模式

vim 的底行工作模式也被称为“最后行模式”，是指可以在接口最底部的一行输入控制操作命令，主要用来进行一些文字编辑的辅助功能，比如字符串搜寻、替代、保存档，以及退出 vim 等。

在命令行工作模式下输入冒号“:”，或者是使用“？”和“/”键，即可以进入底行操作模式，底行工作模式下的常用命令对应的字符串和操作说明如表 3.6 所示。

表 3.6　底行工作模式下的常用命令

命　令	操 作 说 明
q	退出 vim 程序，如果文件有过修改，则必须先保存档
q!	强制退出 vim 而不保存档
x	(exit)保存档并退出 vim
x!	强制保存档并退出 vim
w	(write)保存档，但不退出 vim
w!	对于只读文件，强制保存修改的内容，但不退出 vim
wq	保存档并退出 vim，同 x
E	在 vim 中创建新的档，并可为档命名
N	在本 vim 窗口中打开新的文件
w filename	另存为 filename 文件，不退出 vim
w! filename	强制另存为 filename 文件，不退出 vim
r filename	(read)读入 filename 指定的档内容插入到游标位置

(续表)

命　令	操 作 说 明
set nu	在 vim 的每行开头处显示行号
s/pattern1/pattern2/g	将游标当前行的字符串 pattern1 替换为 pattern2
%s/pattern1/pattern2/g	将所有行的字符串 pattern1 替换为 pattern2
g/pattern1/s//pattern2	将所有行的字符串 pattern1 替换为 pattern2
num1,num2 s/pattern1/pattern2/g	将行 num1 到 num2 的字符串 pattern1 替换为 pattern2
/	查找匹配字符串功能。用“/ 字符串”的命令模式，系统便会自动查找，并突出显示所有找到的字符串，然后转到找到的第一个字符串。如果想继续向下查找，可以按 n 键；向前继续查找则按 N 键
?	也可以使用“? 字符串”查找特定字符串，它的使用与“/ 字符串”相似，但它是向前查找字符串

6. vim 的应用步骤

使用 vim 来编辑一个 C 语言源代码文件的基础应用操作步骤简单总结如下：

(1) 使用“vim+文件名”命令启动 vim 并且创建/打开一个 C 语言文件，此时 vim 位于命令工作模式。

(2) 使用“a”命令进入 vim 的插入工作模式。

(3) 在插入工作模式下对 C 语言源文件的内容进行编辑。

(4) 使用“Esc”键退出 vim 的插入工作模式，进入底行工作模式。

(5) 在底行工作模式下使用“:+wq”命令保存并且退出 vim。

3.2 vim 使用实例

这是一个使用 vim 编写一段应用代码的实例，其详细操作步骤如下：

(1) 使用下列命令打开或者创造一个名称为 Examhello.c 的档，进入如图 3.3 所示的状态。

```
alloeat@ubuntu:~/chapter2Exam$ vim Examhello.c
```

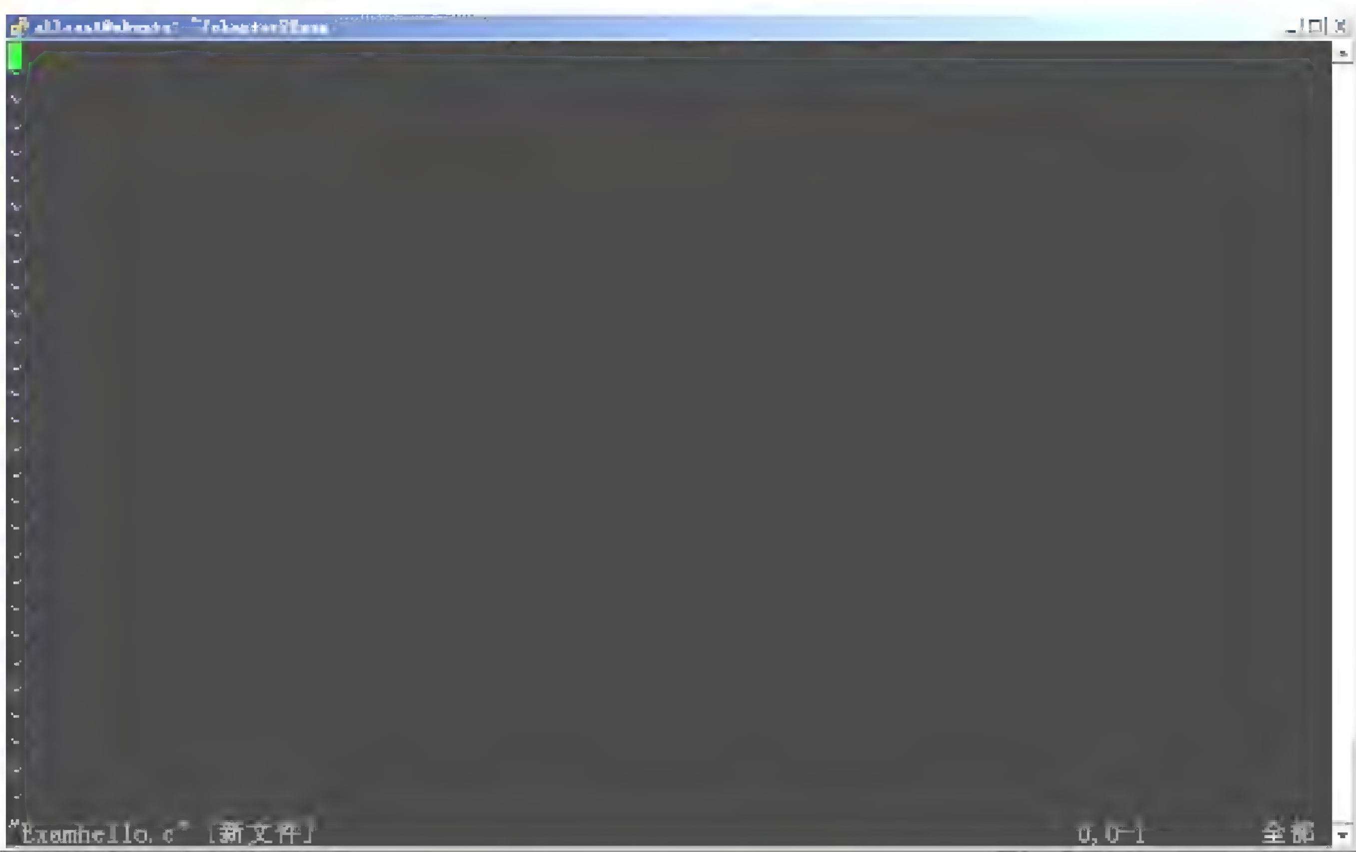


图 3.3　开始编辑状态

(2) 按下 a，进入编辑模式，输入如下的代码，如图 3.4 所示。

```
1  #include   <stdio.h>
2
3  int main(void)
4  {
5      printf("This is a gcc test!\n");
6      return 0;
7  }
```

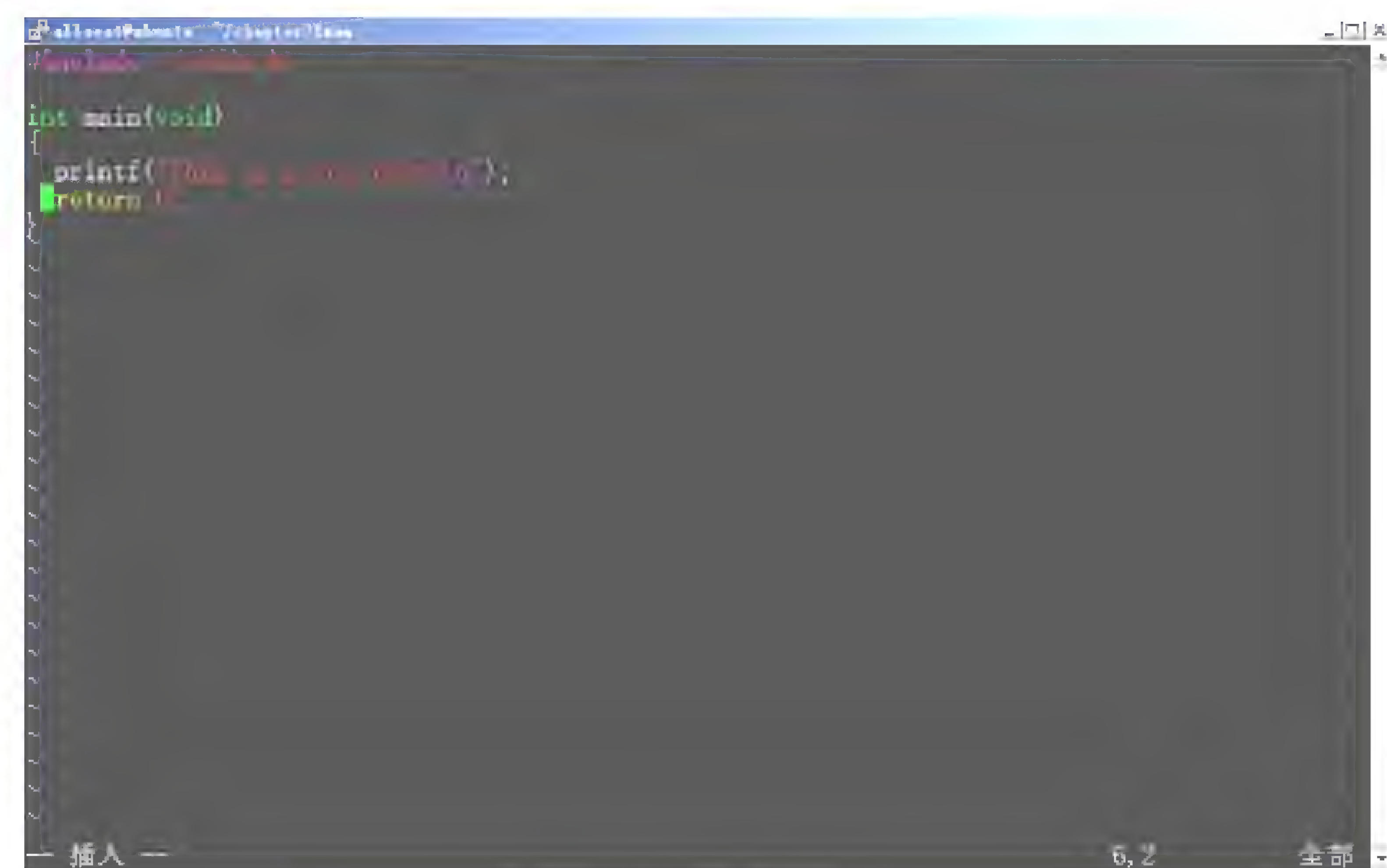


图 3.4　编辑状态

(3) 按下“Esc”键结束编辑，然后使用“;”快捷键进入底行模式，然后输入“wq”保存档并且退出。此时即完成了档的编辑。

注意

在 vim 的实际使用中，有些时候需要对编辑内容中的具体行号位置进行定位，为了方便查找错误对应的行，可以在底行模式使用“set nu”命令在每一行前添加行号，如图 3.5 所示。

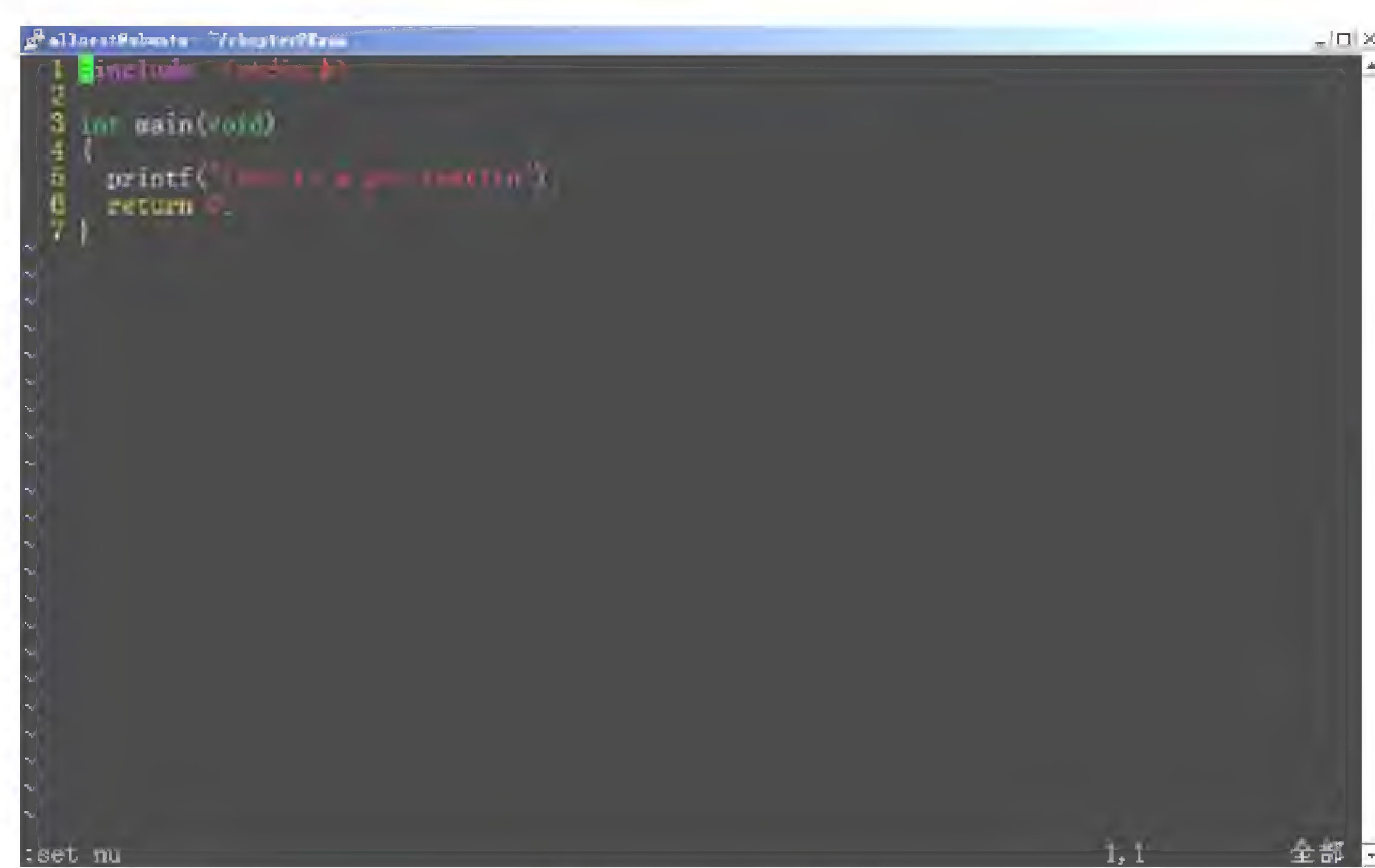


图 3.5　添加行号

3.3 Emacs 的使用

Emacs 是 Linux 下一个功能强大的图形化文本编辑器软件，可以用来编写 C 源程序。Emacs，即 Editor Macros(编辑器宏)的缩写，与 vim 相比，它的一个显著特点是可以使用鼠标进行大部分的操作，对于习惯使用 Windows 系统的用户来说，Emacs 是一个不错的选择。

Emacs 不仅仅是一个文本编辑器，它更是一个整合环境，或称之为集成开发环境。Emacs 是目前世界上最具可移植性的重要软件之一，能够在当前大多数操作系统上运行，包括类 UNIX 系统(GNU/Linux、各种 BSD、Solaris、AIX、IRIX、Mac OS X 等)、MS-DOS、Microsoft Windows 及 OpenVMS 等。

Emacs 既可以在文本终端，也可以在图形用户接口(GUI)环境下运行。使用 GUI 环境下的 Emacs 能够提供菜单(Menubar)、工具栏(toolbar)、滚动条(scrollbar)及上下文菜单(context menu)等交互方式。

Emacs 可以用来编辑文件、收发电子邮件、玩游戏、计算器、浏览网站、查看日历、个人信息管理等。此外，Emacs 支持 Linux 的 Shell 模式，用户可以在 Emacs 中运行 Shell 终端，并在该终端下运行 Shell 命令。也可以直接在 Emacs 的 Shell 模式下运行 Shell 命令。Emacs 还支持对多种编程语言的编译、调试功能，包括 C/C++、Java、Perl、Python、Lisp 等语言。

3.3.1 启动与退出 Emacs

在 Linux 终端命令提示符下使用“emacs”或“emacs filename”命令，即可启动 Emacs 编辑器。也可以在 XWindow 下通过选择“开始”→“编程”→“Emacs”命令进入 Emacs，进入 Emacs 的初始接口如图 3.6 所示。

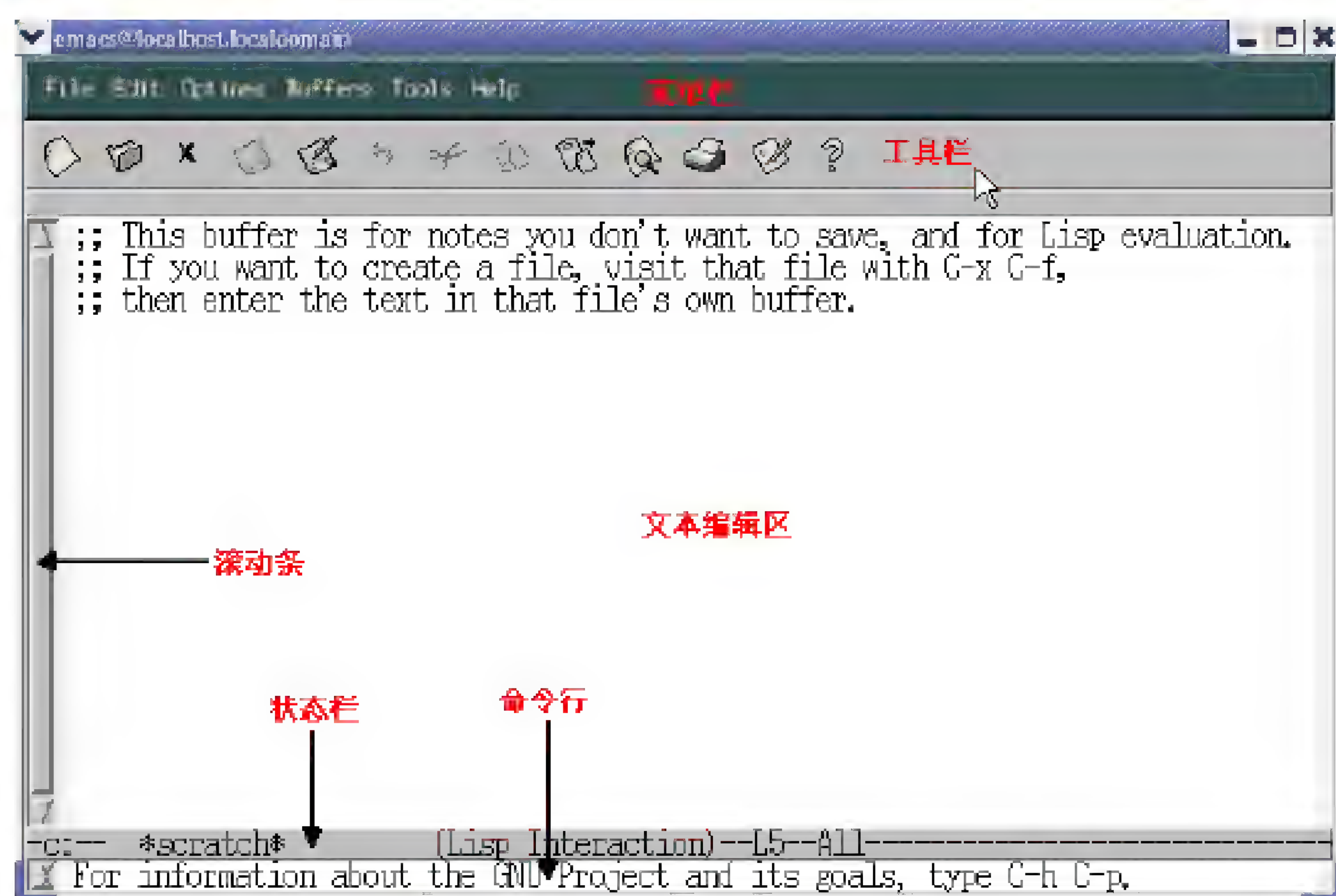


图 3.6　Emacs 的初始界面

要退出 Emacs，直接键入“C-x C-c”即可。

在图 3.6 中，Emacs 的状态区显示了当前文本编辑区载入(运行)的状态。命令行是用户输入命令(不是快捷键！)的区域，比如键入“C-x C-f”后，在这里输入想要打开档的文件名，按“Enter”键后便在文本编辑区打开该档了。

Emacs 的默认工作目录是当前 Linux 用户的主目录，比如在命令行输入“hello”，是指打

开主目录(根用户的主目录是 root)下的 hello 档。

另外，使用 Emacs 编辑器，用户必须了解 Emacs 缓冲区的概念。当用户使用 Emacs 打开或编辑一个档时，Emacs 将会自动创建一个缓冲区(Buffer)，一个档对应一个缓冲区。用户在窗口中进行编辑操作，输入的字符将会被暂存在缓冲区，当用户执行保存操作时，Emacs 会自动将缓冲区中的内容保存到当前打开档中。Emacs 允许用户一次打开多个档，这样就使用了多个缓冲区。

说 明

掌握 Emacs 的快捷键可以说是 Emacs 爱好者的基本功，也是提高编辑速度和质量所必备的，但是初学者可能记不住那么多的快捷键，必要时可以查阅帮助文档或相关书籍，最常用的快捷键数量也就十来个。Emacs 的快捷键都是绑定于“Ctrl”键和“Alt”(或称 Meta)上的，例如“C-x”就是“Ctrl+x”，“M-x”就是“Alt+x”。当然，所有的按键都可以自定义。

3.3.2 Emacs 下的基本操作

本小节中将遇到较多的快捷键的操作，在此有必要先说明键盘操作符号的意义。

- C-x：同时按“Ctrl 键”和“x 键”。
- C x：先按“Ctrl 键”，然后释放它，再按“x 键”。
- M-x：同时按“Alt 键”和“x 键”。
- M x：先按“Alt 键”，然后释放它，再按“x”键。

1. 档操作

选择 Emacs 菜单栏中的“File”命令，在下拉菜单中是一些与文件相关的操作。表 3.7 列出了这些操作中主要的快捷键功能说明。

表 3.7 档操作相关的快捷键

快 捷 键	操 作 说 明
C-x C-f	打开 Emacs 默认目录(用户主目录)下的某个档
C-x d	打开文件路径，将查看某个文件的属性信息，并在这个档上进行编辑操作
C-x i	将某个档的内容插入到当前的缓冲区
C-x C-v	打开一个档，取代当前缓冲区
C-x C-s	保存档
C-x C-w	将当前缓冲区另存为新的档
C-x C-q	切换为只读或者读写模式
C-x C-c	退出 Emacs

2. 编辑操作

选择 Emacs 菜单栏中的“Edit”命令，在下拉菜单中可以看到与文本编辑相关的操作。如表 3.8 所示是这些操作中主要的快捷键功能说明。当然，如果我们不想花时间去记忆这些繁琐

的快捷键，则完全可以使用键盘上的按键，也可以使用下拉菜单中的命令。就像 Windows 下的 Word 软件，我们记忆的快捷键也只有区区几个而已。

表 3.8　编辑操作相关的快捷键

快 捷 键	操 作 说 明	快 捷 键	操 作 说 明
C-f	游标前进一个字符	M->	游标移动到文件尾部
C-b	游标后退一个字符	C-M-f	向前匹配括号
M-f	游标前进一个字	C-M-b	向后匹配括号
M-b	游标后退一个字	C-i	将游标所在位置居中
C-a	游标移动到行首	M-n or C-u n	重复操作随后的命令 n 次
C-e	游标移动到行尾	C-u	重复操作随后的命令 4 次
M-a	游标移动到句首(第一个非空字符)	C-u C-u	重复操作随后的命令 8 次
M-e	游标移动到句尾(最后一个非空字符)	C-x ESC ESC	执行历史命令记录，M-p 选择上一条命令，M-n 选择下一条命令
C-p	光标移动到上一行	C-d	删除一个字符
C-n	光标移动到下一行	M-d	删除一个字
C-v	向下翻页	C-k	删除一行
M-v	向上翻页	M-k	删除一句
M-<	游标移动到文件头部	C-_	撤销操作

3. 窗口操作

窗口就是指 Emacs 的文本编辑区，用户可以使用多个窗口来对同一个缓冲区的不同部分进行操作，比如可以使用“C-x 2”快捷操作使当前编辑区垂直均分为两个窗口；也可以对不同的缓冲区进行操作。如图 3.7 所示为将编辑区垂直均分为两个窗口。

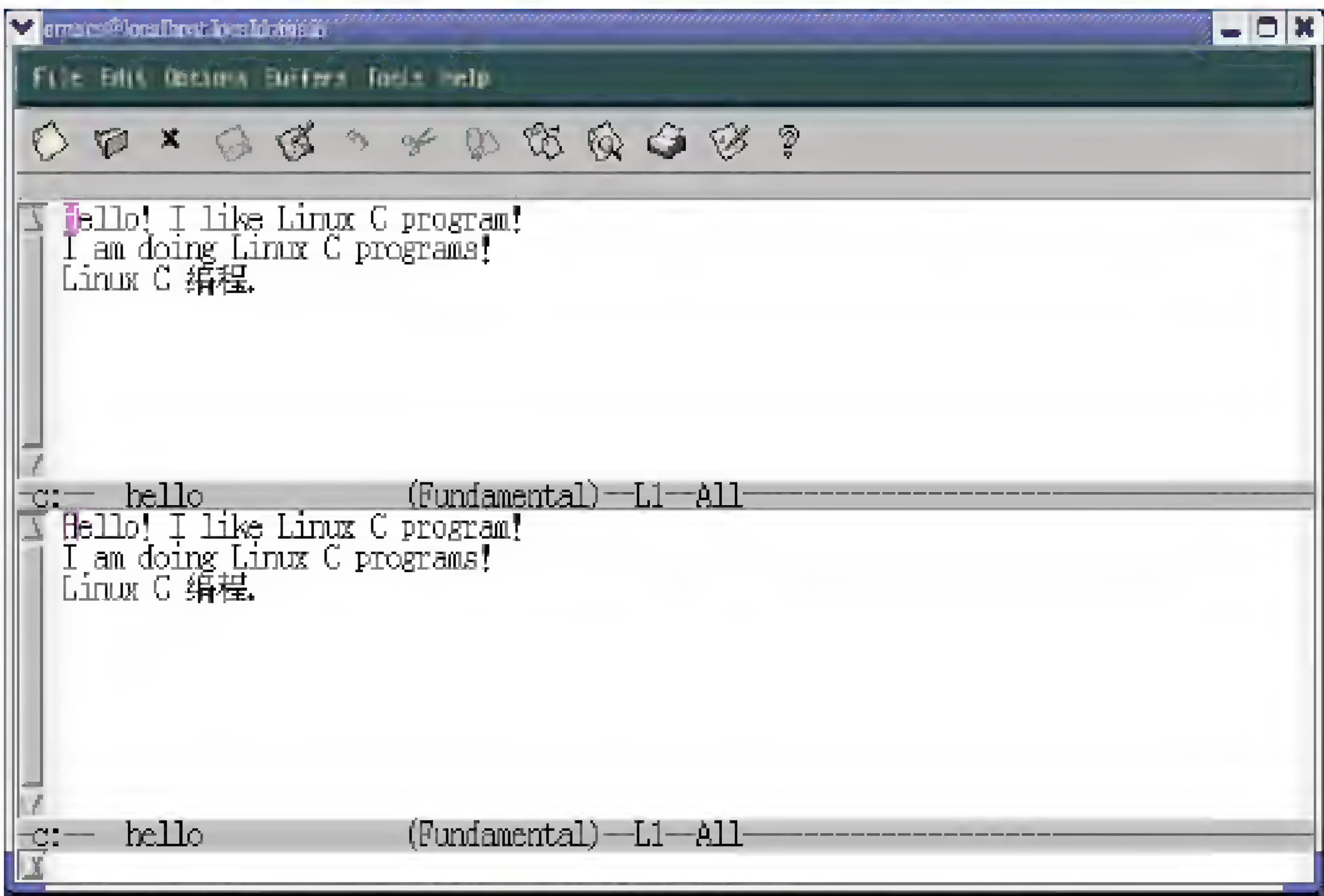


图 3.7　编辑区垂直均分为两个窗口

选择 Emacs 菜单栏中的“Buffers”命令，可以看到在其下拉菜单中列出了当前打开的所有

缓冲区(文件)，用户可以单击不同的文件名，将当前的工作窗口切换至该缓冲区下。当然，也可以使用一些快捷键来对窗口进行操作，它们的说明如表 3.9 所示。

表 3.9 窗口操作相关的快捷键

快 捷 键	操 作 说 明	快 捷 键	操 作 说 明
C-x 0	关闭当前窗口	C-x s	保存所有窗口缓冲
C-x 1	只留下一个窗口	C-x b	选择当前窗口的缓冲区
C-x 2	垂直均分窗口	C-x ^	纵向扩大窗口
C-x 3	水平均分窗口	C-x }	横向扩大窗口
C-x o	切换到其他窗口		

4. 缓冲区列表操作

选择“Buffers”下拉菜单中的“List All Buffers”选项(或使用 C-x C-b)，将在当前窗口打开所有的缓冲区列表。上下移动游标选中不同的行，键入“Enter”后便可对相应的档(缓冲区)进行操作了。在缓冲区列表中的操作快捷方式如表 3.10 所示。

表 3.10 缓冲区列表的操作

快 捷 键	操 作 说 明	快 捷 键	操 作 说 明
C-x C-b	打开缓冲区列表	u	取消标记
d or k	标记为删除	x	执行标记的操作
~	标记为未修改状态	f	在当前窗口打开该缓冲区
%	标记为只读	o	在其他窗口打开该缓冲区
s	保存缓冲		

5. 程序编译

Emacs 可以支持多种编程语言的编辑模式，例如 C、C++、Java 等语言。用户可以通过键入“M-x [language]-mode”命令(M-x 是快捷键操作，[language]-mode 是命令行输入的命令)来选择各种不同语言模式的编辑环境，[language]表示不同的编程语言。

例如我们首先键入“M-x”快捷方式，进入 Emacs 的程序编译模式，然后在命令行区域输入“c-mode”，按“Enter”键后将出现如图 3.8 所示的接口。

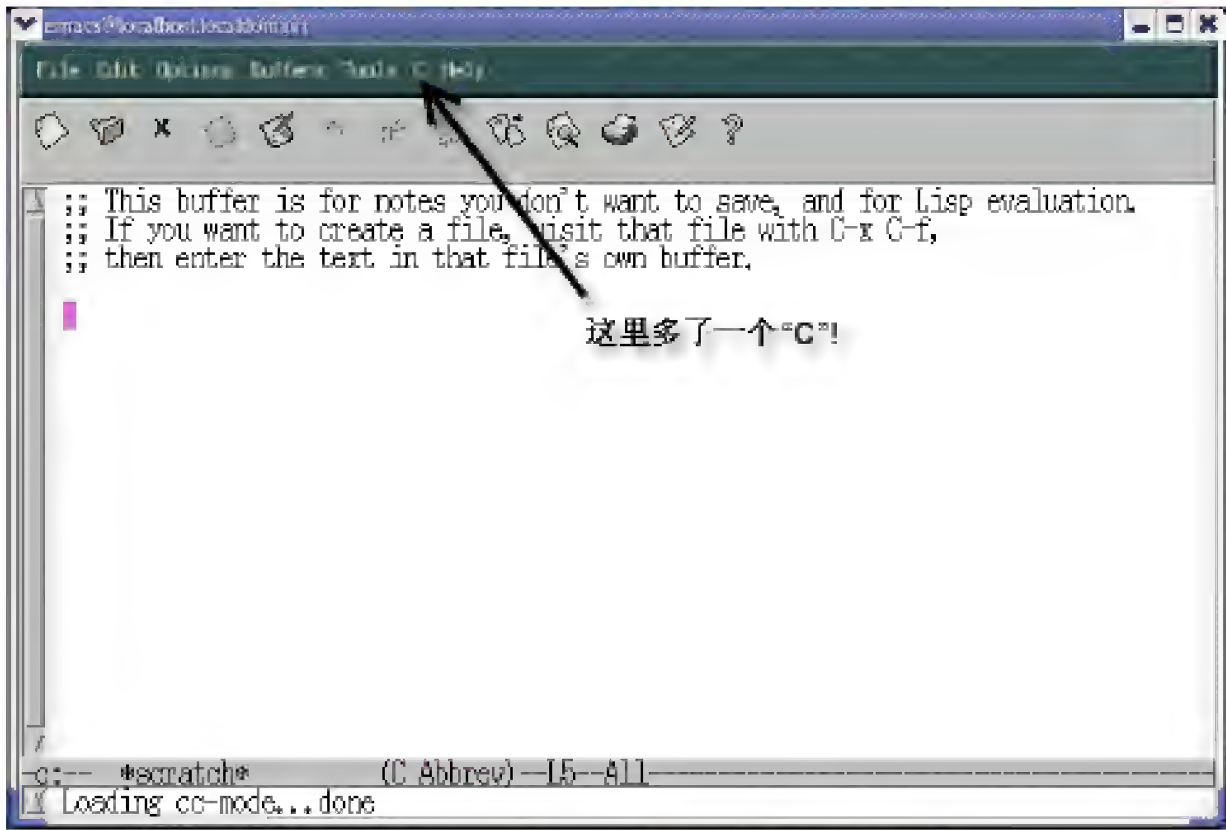


图 3.8 Emacs 的 C 编辑模式

可以看到，在图 3.8 中，编辑器的菜单栏中多了一个“C”选项，表明 Emacs 编辑区进入了 C 语言的编辑模式。读者可以试试输入“c++-mode”或者“Java-mode”命令时，编辑器的菜单栏将会发生怎样的变化。

在程序编译模式下，当用户输入代码时，Emacs 支持自动缩进显示。事实上，Emacs 可以支持多种缩进风格，在 C 模式下，用户可以通过使用“M-x c-set-style”命令来选择自己需要的缩进风格。

Emacs 不仅仅是一个编辑器，更是一个集成开发环境，可以使用它来进行 C 程序(当然也可以是其他的程序设计语言)的编译和调试。用户在“Tools”菜单中找到“Compile”选项，或者直接键入“M-x compile”，就可以在 Emacs 的命令行输入编译命令了。

如果有 Makefile 档(将在第 5 章向读者介绍)，就接受默认设置，使用“make -k”命令来编译程序。编译出现错误和警告时，程序员可以单击鼠标来定位这些警告和错误。

出现了警告和错误信息，就离不开调试这一重要步骤。Emacs 还支持程序的调试功能，用户可以使用命令“M-x gdb”来调用 Linux 下的 gdb 调试器，或者在 Tools 菜单中选择 gdb 选项，然后即可输入调试命令。

表 3.11 列出了在程序编译模式下的常用快捷操作。

表 3.11　程序编译模式下的操作

快　捷　键	操 作 说 明	快　捷　键	操 作 说 明
M-x compile	执行编译操作	M-x xdb	调用 xdb 调试器
M-x gdb	调用 gdb 调试器	M-x sdb	调用 sdb 调试器
M-x dbx	调用 dbx 调试器		

在表 3.11 中，gdb、dbx、xdb 和 sdb 是 Linux 下的各种程序调试工具(将在第 4 章中向读者介绍)，当用户输入不同的命令时，Emacs 便会自动调用 Linux 下的这些调试器来对当前缓冲区中的程序进行调试。所以，与其说 Emacs 是一个文本编辑区，不如说它更像是一个功能强大的集成开发环境。

6. 搜索模式

Emacs 支持对当前窗口文件中的字符搜索功能，这无疑会使 Emacs 下的文本编辑工作变得更加方便、适用。Emacs 的字符搜索相关的快捷键操作如表 3.12 所示。

表 3.12　字符搜索操作

快 捷 键	操 作 说 明
C-s 字符	向前搜索字符，查找到的字符以蓝色字体显示
ENTER	停止搜索
C-r 字符	向后搜索字符，查找到的字符以蓝色字体显示
C-s C-w	以游标所在位置的字为关键词进行搜索
C-s C-s	重复上一次搜索
C-r C-r	重复上一次反向搜索

(续表)

快 捷 键	操 作 说 明
C-s ENTER C-w	进入单词搜索模式，搜索完毕后，游标停留在查找到的第一个单词的后面
C-r ENTER C-w	进入反向单词搜索模式
C-r	在进入查找/替换模式后，该命令进入迭代编辑模式
C-M-x	退出迭代编辑模式，返回到查找/替换模式

7. Shell 模式

Emacs 编辑器最显著的特点之一是它支持 Linux 的 Shell 模式，用户可以在 Emacs 的文本编辑区运行 Shell 终端，并在该终端下运行 Shell 命令。比如键入“M-x”快捷键后，在 Emacs 的命令行输入“shell”，按“Enter”键后，Emacs 便会在当前窗口打开一个 Shell 终端。我们在 Shell 终端中运行命令，如图 3.9 所示。

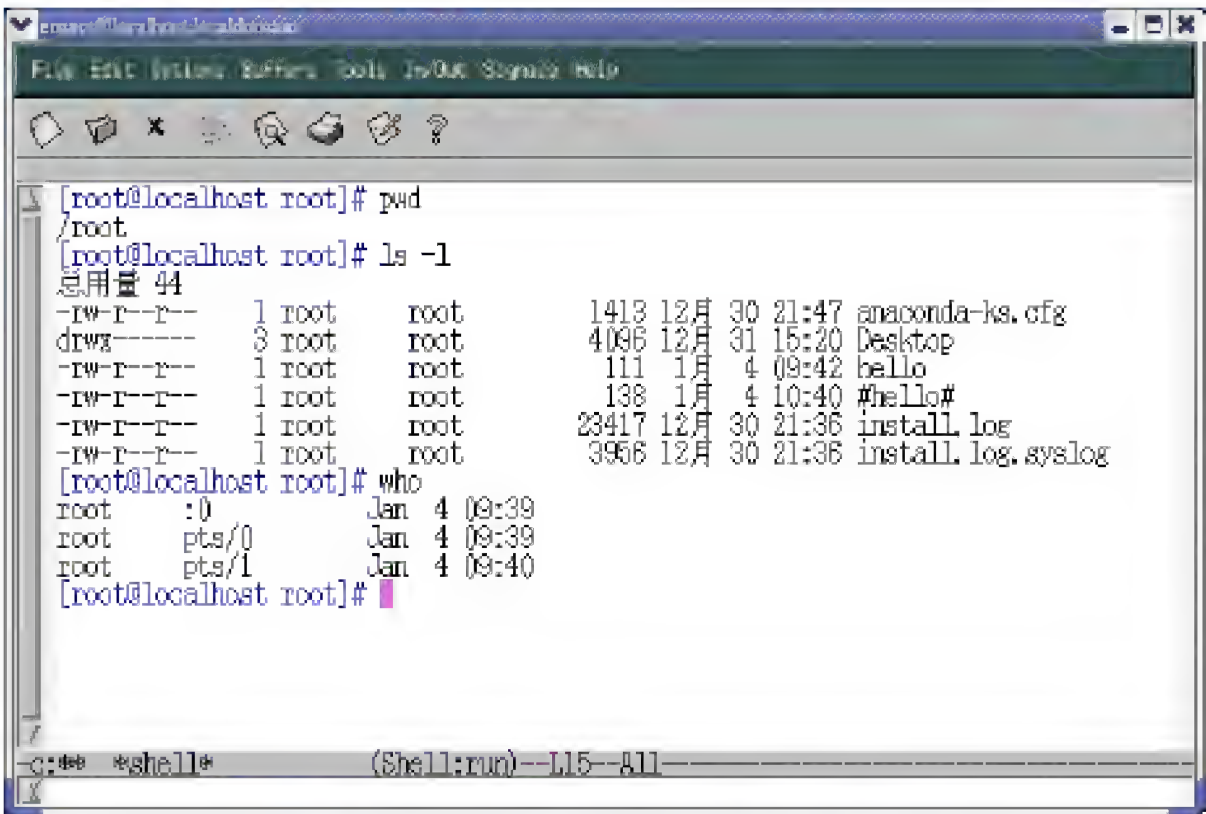


图 3.9　Emacs 下运行 Shell

另外，也可以直接在 Emacs 的命令行中执行 Linux Shell 的任何命令，并将执行结果输出在文本编辑区。选择菜单栏中“Tools”选项下的“Shell command”或键入“M-!”，即可进入 Shell 模式。

例如，键入“M-!”后，Emacs 进入 Shell 模式，此时在 Emacs 的命令行输入“ls -l”命令，按“Enter”键后编辑器的接口如图 3.10 所示。

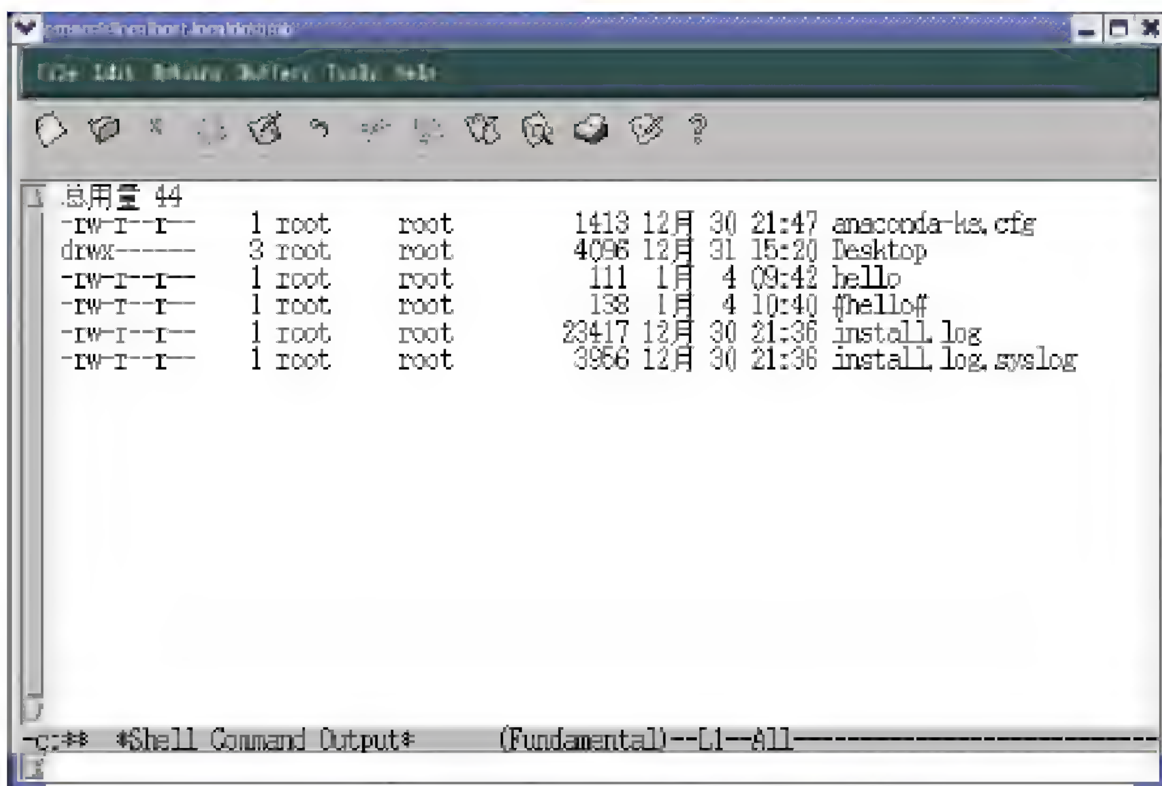


图 3.10　Emacs 中执行 Shell 命令

在图中可以看到，“ls -l”命令的执行结果显示在 Emacs 的文本编辑区。由于 Shell 命令的

输出是在一个编辑缓冲区里，因此我们可以对它进行编辑、保存等操作。所以，在想要保存Linux 某一个 Shell 命令的执行结果的场合中，Emacs 的这种模式就显得十分适用。

表 3.13 列出了在 Emacs 的 Shell 模式下常用的快捷键操作说明。

表 3.13　执行 Shell 命令的快捷键

快　捷　键	操 作 说 明
M-x shell	打开 Shell
M-!	执行 Shell 命令(Shell-command)
M-1 M-!	执行 Shell 命令，命令的输出插入在游标当前位置，而不打开新的输出窗口
M-|	针对某一特定区域执行 Shell 命令(Shell-command-on-region)
M-! M-p	执行前一条 Shell 命令，同 M-!+向上方向键
M-! M-n	执行下一条 Shell 命令，同 M-!+向下方向键

此外，Emacs 还具有很多其他的功能，比如收发电子邮件、玩游戏、计算器、浏览网站、查看日历、个人信息管理等，鉴于篇幅和本书的介绍范围，在此不一一列举，用户也可以查看Emacs 的帮助手册来获得更多的信息。

3.4 Emacs 使用实例

本节以 3.2 节中使用 vim 编辑好的 vim_test.c 档为例，在 Emacs 中编译并运行该程序，并将程序的运行结果保存到文本文件 vim_test_result 中，主要向读者演示了 Emacs 的 C 程序编译模式与 Shell 模式下的常见操作。我们将操作步骤详细描述如下：

(1) 在 Linux 命令行下输入“emacs vim_test.c”命令，使用 Emacs 编辑器打开 vim_test.c 文件。

(2) 键入“M-x c-mode”命令，进入 Emacs 的 C 程序编译模式，如图 3.11 所示。

(3) 键入“M-x compile”，此时便可以在 Emacs 的命令行输入 C 程序编译命令了，这里输入编译命令“gcc -o vim_test vim_test.c”，按“Enter”键后程序的编译结果如图 3.12 所示。

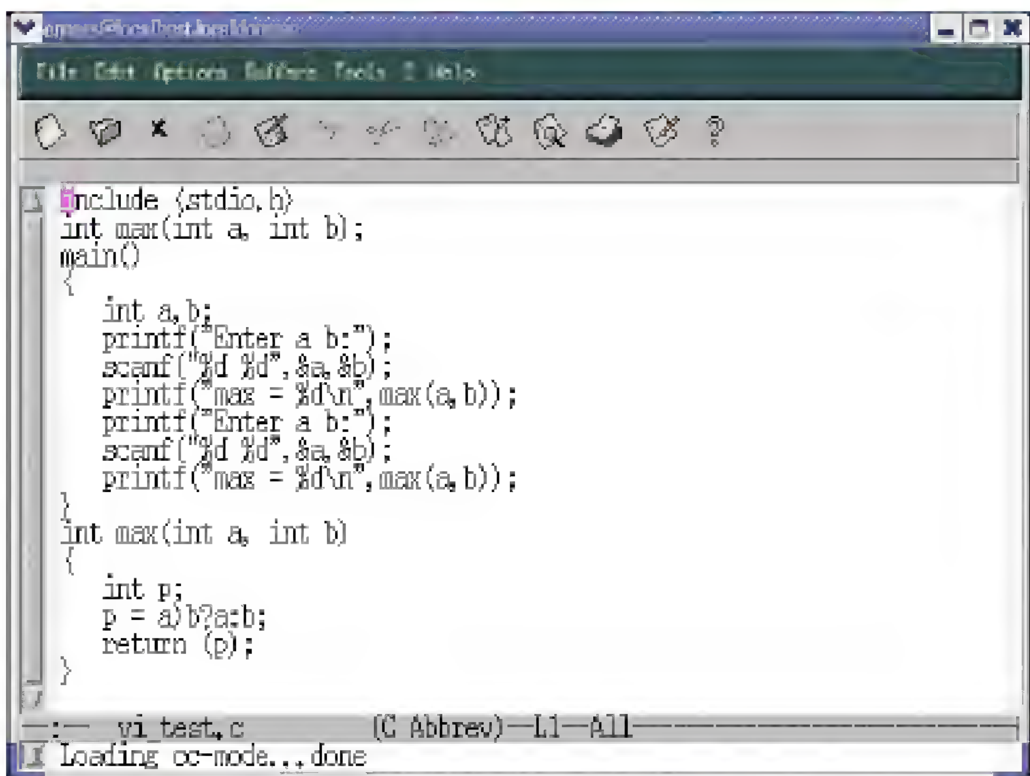


图 3.11　vim_test.c 的编译模式

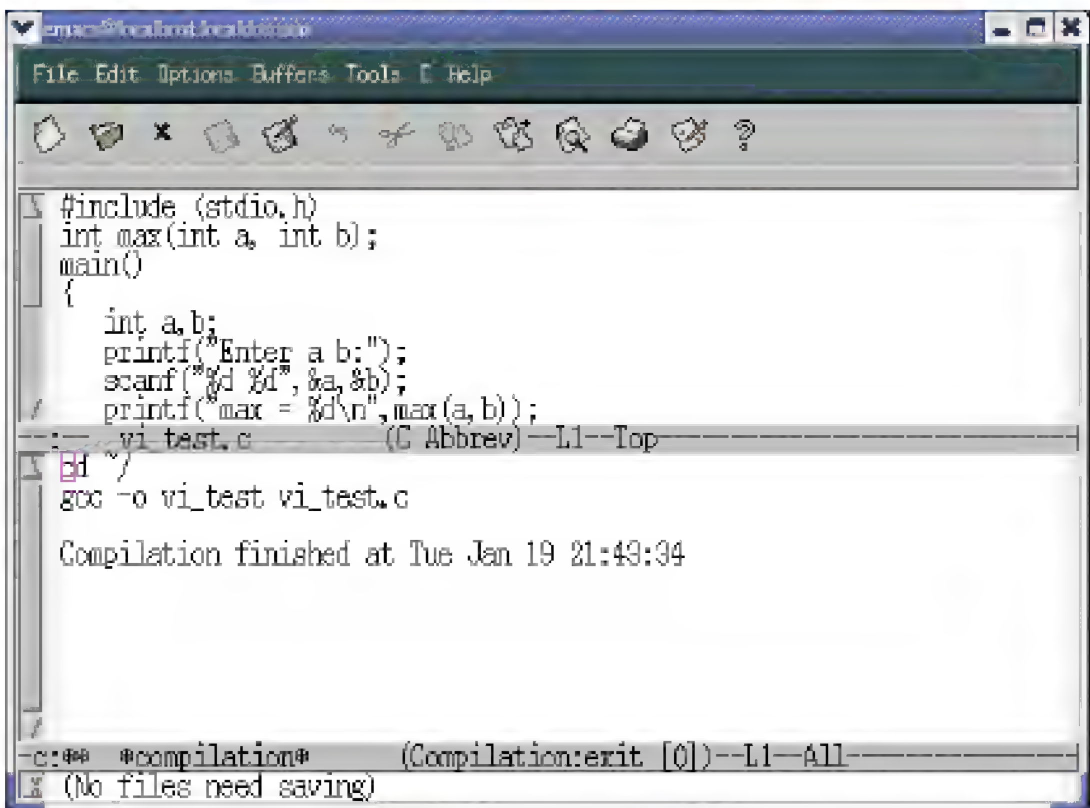


图 3.12　调用 gcc 编译 vim_test.c

(4) 将游标定位在编辑区的下面一个窗口，然后键入“C-x 0”关闭当前窗口。

(5) 键入“M-x shell”，按“Enter”键后 Emacs 自动在当前窗口打开一个 Shell 终端。

(6) 在该 Shell 终端中运行编译通过的可执行程序 vim_test，如图 3.13 所示。

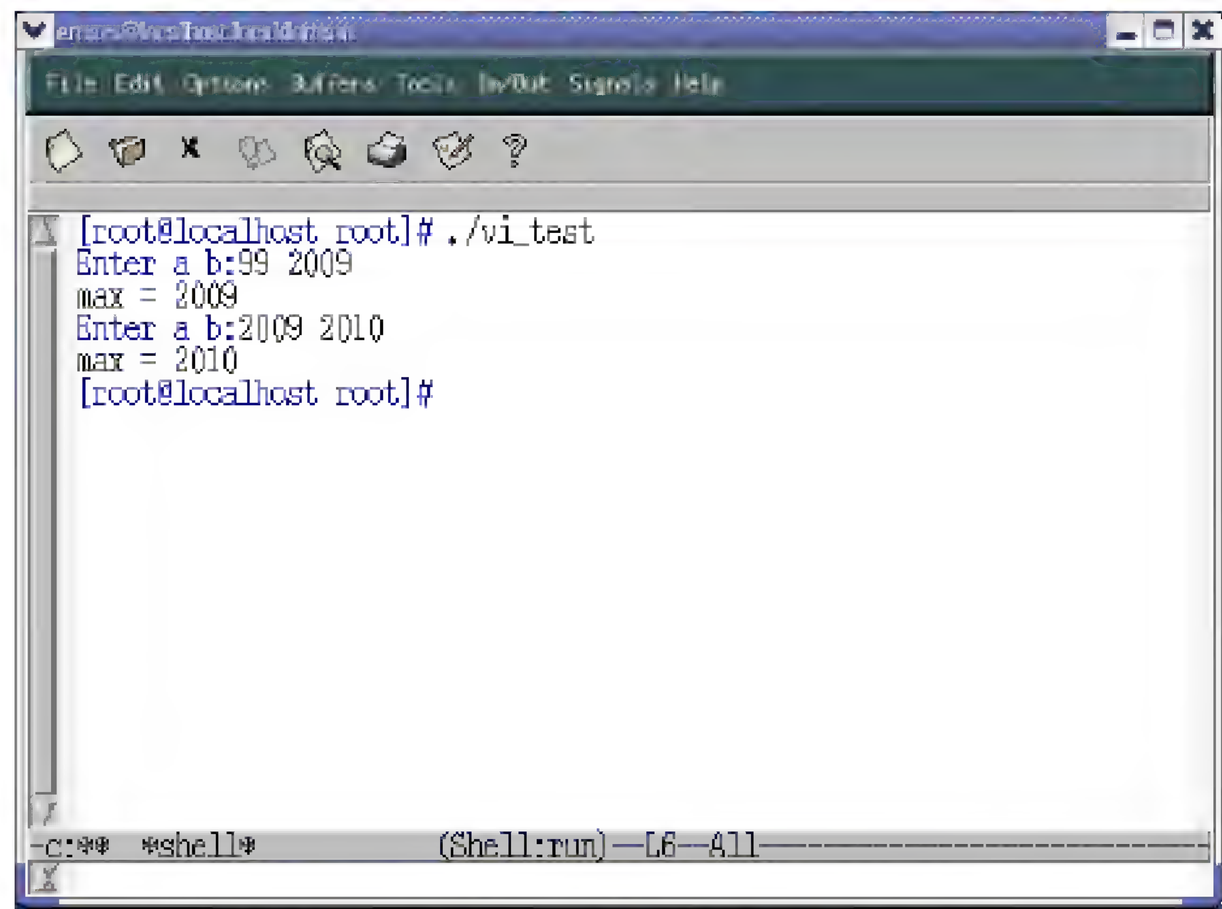


图 3.13　调用 Shell 执行 vi_test 程序

(7) 键入“C-x C-w”，将当前缓冲区的内容保存为新的档，我们在命令行输入想要保存的文件名 vim_test_result，按“Enter”键后便将 vim_test 程序的执行结果保存到 vim_test_result 文本文件中去了。

(8) 键入“C-x C-c”，退出 Emacs。

3.5 本章小结

本章向读者介绍了 Linux 下最常见的两种编辑器 vim 和 Emacs，它们不仅是文本编辑器，也是进行程序开发的环境，因此熟练掌握和应用这两种编辑器，尤其是它们各种工作模式下的命令及快捷操作，是进行 Linux 下 C 程序开发的基本本领。

实战演练

1. 使用 vim 编辑器编辑下面这段文字：

```
Hello! I like Linux C program!
I am doing Linux C programs!
Linux C  编程.
```

保存为档 hello.txt，然后退出 vim。

2. 验证 vim 编辑器 3 种工作模式间的切换命令。

3. 验证当使用“A”、“I”、“O”命令将 vim 从命令行模式切换至插入模式时，分别与“a”、“i”、“o”有何不同?

4. 试用 vim 打开/usr/src/linux-x.x.xx/kernel(x.x.xx 表示 Linux 的内核版本号，不同用户会有

所不同)目录下的 fork.c 文件，在 vim 的命令行模式下复制某一行的内容，粘贴至下一行，然后再取消当前的操作，并强制退出。

5. 用 vim 打开/usr/src/linux-x.x.xx/kernel 目录下的 fork.c 文件，查找该文件中的字符串“fork”，使其以红色字体显示。

6. 在 Emacs 中打开一个 Shell 终端，执行 ls -l 命令，并将命令执行的结果保存到档 ls_result.txt 中。

7. 用 Emacs 编写一个 Hello World 的程序，保存为 hello.c 文件，并编译运行该程序。程序如下：

```
#include <stdio.h>
main()
{
    printf("Hello, Linux World!\n");
}
```

8. 用 Emacs 打开 hello.c 文件，将当前窗口均分为 4 个工作窗口，在不同的窗口对同一文件的不同部分进行编辑。

9. 用 Emacs 打开/root 目录下的 install.log 文件，查找该文件中的字符“i”，使其以蓝色字体显示。

10. 在 Emacs 中执行 Shell 命令，比如 ls、pwd、who、uname 等。

第 4 章

gcc编译器与gdb调试器

gcc 编译器是 GNU 开源组织发布的 UNIX/Linux 下功能强大、性能优越的多平台编译器，它可以将 C、C++等多种语言编写的源程序编译、链接成可执行文件。而 gdb 是 GNU 推出的功能强大的程序调试器，可以说 gcc 与 gdb 是在 Linux 环境下进行 C 程序开发不可或缺的工具，也是 Linux 程序员必须掌握的技能之一。本章将向读者介绍 gcc 编译器与 gdb 调试器。



本章内容：

◎ gcc 编译器简介。
◎ gcc 使用详解。
◎ gdb 调试器简介。
◎ gdb 使用详解。
◎ 其他类型的调试器简介。

4.1 gcc 编译器简介

在 Linux 环境下开发应用程序时，绝大多数情况下使用的都是 C 语言，因此，几乎每一位 Linux 程序员面临的首要问题都是如何灵活运用 C 编译器。目前 Linux 下最常用的 C 语言编译器是 gcc(GNU Compiler Collection)，它是 GNU 项目中符合 ANSI C 标准的编译系统，能够编译用 C、C++和 Object C 等语言编写的程序。事实上，gcc 可以编译如 C、C++、Object C、Java、Fortran、Pascal、Modula-3 和 Ada 等多种语言，而且 gcc 又是一个交叉平台编译器，它能够在当前 CPU 平台上为多种不同体系结构的硬件平台开发软件，因此尤其适合在嵌入式软件领域的开发编译。在使用 gcc 编译程序时，编译过程可以被细分为 4 个阶段：

- 预处理(Pre-Processing)。
- 编译(Compiling)。
- 汇编(Assembling)。
- 链接(Linking)。

Linux 程序员可以根据自己的需要让 gcc 在编译的任何阶段结束，以便检查或使用编译器在该阶段的输出信息，或者对最后生成的二进制文件进行控制，以便通过加入不同数量和种类的调试代码来为今后的调试做好准备。和其他常用的编译器一样，gcc 也提供了灵活而强大的代码优化功能，利用它可以生成执行效率更高的代码。

gcc 提供了 30 多条警告信息和 3 个警告级别，使用它们有助于增强程序的稳定性和可移植性。此外，gcc 还对标准的 C 和 C++语言进行了大量的扩展，提高程序的执行效率，有助于编译器进行代码优化，能够减轻编程的工作量。另外，gcc 通过文件后缀名来区别输入文件的类别，表 4.1 介绍了 gcc 所遵循的部分约定规则。

表 4.1 gcc 的文件类型约定规则

文件后缀名	文件类型约定
.c	C 语言源代码文件
.a	由目标文件构成的档案库文件
.C、.cc 或.cxx	C++源代码文件
.h	程序所包含的头文件
.i	经预处理过的 C 源代码文件
.ii	经预处理过的 C++源代码文件
.m	Objective-C 源代码文件
.o	编译后的目标文件
.s	汇编语言源代码文件
.S	经过预编译的汇编语言源代码文件

gcc 作为 Linux 下 C/C++重要的编译环境，功能强大，编译选项繁多。为方便本章后续内容的讲解，在此先将 gcc 的常用编译选项在表 4.2 中列出。通常情况下，这些选项是多个联合在一起使用的，掌握它们的含义和使用技巧是一个优秀的 Linux 程序员所应该具备的能力。

表 4.2 gcc 常用选项

选　项	含 义 描 述
-o filename	指定输出文件名，在编译为目标代码时，这一选项不是必需的。如果 filename 没有指定，默认文件名是 a.out
-c	只编译不链接，生成目标文件“.o”
-S	只编译不汇编，生成汇编代码
-E	只进行预编译，不做其他处理
-g	在生成的可执行程序中包含标准调试信息
-v	打印编译器内部编译各过程的命令行信息和编译器的版本号
-I dir	在头文件的搜索路径列表中添加 dir 目录
-L dir	在库文件的搜索路径列表中添加 dir 目录
-static	链接静态库
-library	链接名为 library 的库文件
-Dmacro	定义指定的宏，使它能够通过源码中的#ifdef 进行检验
-O、-O2、-O3	将优化状态打开，该选项不能与-g 选项联合使用
-Wall	在发生警告时取消编译操作，即将警告看作是错误
-Werror	在发生警告时取消编译操作，即将警告看作是错误
-w	禁止所有的报警
-pedantic	严格要求符合 ANSI 标准

说 明

gcc 编译的选项能够分辨大小写，所以使用时要特别注意。

4.2 如何使用 gcc

gcc 编译器的功能强大，包括警告提示功能、代码优化、连接库、使用管道加速等，下面通过各个具体的实例向读者一一介绍 gcc 的这些功能，以及其他的一些知识点。通过本节的学习，读者应该能够熟练应用 gcc 来编译 C/C++程序。

4.2.1 安装和配置 gcc

在 Ubuntu 12.04 中 gcc 是默认安装的，但是其还缺少常用的头文件和库文件，所以还需要

安装 build-essential 这个包，可以在联网状态下使用如下命令来安装这个包。

```
# sudo apt-get install build-essential
```

其中，apt-get 是 Ubuntu 下的软件管理命令，它可以安装、删除、更新系统中的软件包。install 是安装软件包，build-essential 是待安装的软件包名称。由于安装软件需要 root 权限，因此，系统会提示输入密码，在输入密码后，系统会自动安装编译所需要的相关文件。系统在安装 build-essential 时，会把程序文件放入以下几个目录：

➢ /usr/lib

大部分的编译程序放在这个目录。在这里由编译时需要的可执行程序，还有一些特定版本的库文件与头文件等。

➢ /usr/bin/gcc

该目录指的是编译程序，即实际在命令行中执行的程序。这个目录可供各个版本的 gcc 使用，只要用不同的编译程序目录来安装就可以。

➢ /usr/include

这个目录及其子目录下包含程序所需要的头文件。缺少头文件，gcc 在编译时会出现找不到头文件的错误。

在安装完成之后，可以使用“gcc-v”命令来查看 gcc 的版本号：

```
# gcc -v
使用内建 specs。
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/i686-linux-gnu/4.6/lto-wrapper
目标：i686-linux-gnu
配置为：../src/configure -v --with-pkgversion='Ubuntu/Linaro 4.6.3-1ubuntu5'
--with-bugurl=file:///usr/share/doc/gcc-4.6/README.Bugs --enable-languages=c,c++,fortran,objc,obj-c++ --prefix=/usr
--program-suffix=-4.6 --enable-shared --enable-linker-build-id --with-system-zlib --libexecdir=/usr/lib
--without-included-gettext --enable-threads=posix --with-gxx-include-dir=/usr/include/c++/4.6 --libdir=/usr/lib
--enable-nls --with-sysroot=/ --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes
--enable-gnu-unique-object --enable-plugin --enable-objc-gc --enable-targets=all --disable-werror --with-arch-32=i686
--with-tune=generic --enable-checking=release --build=i686-linux-gnu --host=i686-linux-gnu --target=i686-linux-gnu
线程模型：posix
gcc 版本 4.6.3 (Ubuntu/Linaro 4.6.3-1ubuntu5)
```

注意

由于 gcc 仍然处于不断地完善与更新之中，每隔几个月就会有新的稳定发行版本产生，用户可以通过访问 http://www.gnu.org/software/gcc/来了解 gcc 的最近发展，下载最新的软件套件。

4.2.2 gcc 编译初步

如前所述，gcc 的编译过程分为预处理、编译、汇编、链接 4 个阶段。从功能上分，预处理、编译和汇编是 3 个不同的阶段，但 gcc 在实际操作上，可以把这 3 个步骤合并为一个步骤

来执行。下面通过简单而通用的 Hello World 的例子来向读者讲解 gcc 的工作原理。首先用熟悉的编辑器 vim 或者是 Emacs 输入程序 4.1 所示的代码，并保存文件名为 hello.c。

【程序 4.1】Hello World 的例子：hello.c。

```
#include <stdio.h>
int main(void)
{
printf ("hello world, Linux programming!\n");
    return 0;
}
```

然后输入下面的命令编译和运行这段程序：

```
# gcc hello.c
# ./a.out
```

程序输出结果：

```
hello world, Linux programming!
```

运行上述的 gcc 编译命令后，编译器自动创建了一个名为 a.out 的输出文件，而“./”表示执行当前目录下的可执行程序或脚本程序。当然，用户可以使用 gcc 选项-o 来改变编译后的文件名，如使用下面的命令，将输出文件命名为 hello，执行 hello 文件后得到相同的结果。

```
# gcc -o hello.c hello
# ./hello
hello world, Linux programming!
```

说 明

书中的代码信息中，在命令提示符“#”后的文字属于用户输入的 Linux 的操作命令，而其他属于程序运行的结果或者是系统产生的信息。

现在回到程序 4.1 的编译过程，从程序员的角度看，只需简单地执行一条 gcc 命令就可以了，但从编译器的角度来看，却需要完成一系列非常繁杂的工作。首先，gcc 需要调用预处理程序 cpp，由它负责展开在源文件中定义的宏，并向其中插入“#include”语句所包含的内容；接着，gcc 会调用 ccl 和 as 将处理后的源代码编译成目标代码；最后，gcc 会调用链接程序 ld，把生成的目标代码链接成一个可执行程序。

为了更好地理解 gcc 的工作过程，可以把上述编译过程分成几个步骤单独进行，并观察每一步的运行结果。

(1) 预处理。在预处理阶段，gcc 把预处理命令扫描处理完毕，输入 C 语言的源文件，通常为*.c，它们通常带有.h 之类头文件的包含文件。这个阶段主要处理源文件中的#ifdef、#include 和#define 等预处理命令。该阶段会生成一个中间文件*.i，但实际工作中通常不会专门生成这种文件，因为基本上用不到，若非要生成这种文件，可以使用-E 参数让 gcc 在预处理结束后停止编译过程：

```
# gcc -E hello.c -o hello.i
```

此时若查看 hello.i 文件中的内容，会发现头文件 stdio.h 的内容确实都插入到文件里去了，而其他应当被预处理的宏定义也都做了相应的处理。

(2) 编译阶段。在编译阶段，gcc 把预处理后的结果编译成汇编或者目标模块。输入的是中间文件*.i，编译后生成汇编语言文件*.s。这个阶段对应的 gcc 命令如下所示：

```
gcc -S hello.i -o hello.s
```

(3) 汇编。在汇编阶段，编译器把编译出来的结果汇编成具体 CPU 上的目标代码模块。输入汇编文件*.s，输出机器语言*.o。这个阶段可以通过使用-c 参数来完成：

```
# gcc -c hello.s -o hello.o
```

(4) 在链接阶段把多个目标代码模块连接生成一个大的目标模块。输入机器代码文件*.o(与其他的机器代码文件和库文件)，汇集成一个可执行的二进制代码文件。这一步骤可以利用下面的示例命令完成：

```
# gcc hello.o -o hello
```

到这里 gcc 就完成了整个编译过程，得到可执行文件。

4.2.3 警告提示功能

gcc 包含完整的出错检查和警告提示功能，它们可以帮助 Linux 程序员写出更加专业和优美的代码。先来阅读程序 4.2 所示的代码，这段代码写得很糟糕，仔细检查一下会挑出很多毛病：

- main 函数的返回值类型被声明为 void，但实际上应该是 int。
- 使用了 GNU 语法扩展，即使用 long long 来声明 64 位整数，不符合 ANSI/ISO C 语言标准。
- main 函数在终止前没有调用 return 语句。

【程序 4.2】让 gcc 产生告警的代码：warning_code.c。

```
#include <stdio.h>
void main(void)
{
long long int var=2010;
printf("It is a warning code for gcc!\n");
}
```

下面来看看 gcc 是如何帮助程序员来发现这些错误的。当 gcc 在编译不符合 ANSI/ISO C 语言标准的源代码时，如果加上了-pedantic 选项，那么在使用了扩展语法的地方将产生相应的警告信息，如下：

```
# gcc -pedantic warning_code.c -o warning_code
warning_code.c: In function `main':
warning_code.c: 4: warning: ISO C89 does not support `long long'
warning_code.c: 3: warning: return type of `main' is not `int'
```

现在我们对程序 4.2 的代码进行修改，将 main 函数的返回值声明为 int 型，var 变量定义为长整型(long int)，在 main 函数的最后增加返回语句“return 0;”，如程序 4.3 所示。

【程序 4.3】标准的代码：standard_code.c。

```
#include <stdio.h>
int main(void)
{
    long int var = 2010;
    printf("It is a standard code for gcc!\n");
    return 0;
}
```

然后再次使用 gcc 的-pedantic 选项进行编译：

```
# gcc -pedantic standard_code.c -o standard_code
```

可以看到，此时就没有任何警告信息了。

需要注意的是，-pedantic 编译选项并不能保证被编译程序与 ANSI/ISO C 标准的完全兼容，它只能用来帮助 Linux 程序员离这个目标越来越近，换句话说，-pedantic 选项能够帮助程序员发现一些不符合 ANSI/ISO C 标准的代码，但不是全部。事实上只有 ANSI/ISO C 语言标准中要求进行编译器诊断的那些情况，才有可能被 gcc 发现并提出警告。

除了-pedantic 之外，gcc 还有一些其他编译选项也能够产生有用的警告信息。这些选项大多以-W 开头，其中最有价值的是-Wall，使用它能够使 gcc 产生尽可能多的警告信息，下面使用-Wall 选项再次编译 warning_code.c：

```
# gcc -Wall warning_code.c -o warning_code
warning_code.c: In function `main':
warning_code.c: 3: warning: return type of `main' is not `int'
warning_code.c: 4: warning: unused variable `var'
```

gcc 给出的警告信息虽然从严格意义上说不能算错误，但却很可能成为错误的栖身之所。一个优秀的 Linux 程序员应该尽量避免产生警告信息，使自己的代码始终保持简洁、优美和健壮的特性。

在处理警告方面，另一个常用的编译选项是-Werror，它要求 gcc 将所有的警告当成错误进行处理，这在使用自动编译工具(如 Make 等)时非常有用。如果编译时带上-Werror 选项，那么 gcc 会在所有产生警告的地方停止编译，迫使程序员对自己的代码进行修改。只有当相应的警告信息消除时，才可能将编译过程继续朝前推进。下面使用-Werror 选项再次编译 warning_code.c：

```
# gcc -Werror warning_code.c -o warning_code
cc1: warnings being treated as errors
warning_code.c: 3: warning: return type of `main' is not `int'
warning_code.c: In function `main'
warning_code.c: 4: warning: unused variable `var'
```

对 Linux 程序员来讲，gcc 给出的警告信息是很有价值的，它们不仅可以帮助程序员写出更加健壮的程序，而且还是跟踪和调试程序的有力工具。建议在用 gcc 编译源代码时始终带上-Wall 选项，并把它逐渐培养成一种习惯，这对找出常见的隐式编程错误是很有帮助的。

4.2.4 优化 gcc

代码的优化功能是现代 C 编译器一个最令人心动的特性，作为 Linux 平台下的标准 C 编译器，gcc 拥有强大并且是可配置的优化器，从而能够对程序进行优化处理。

代码优化指的是编译器通过分析源代码，找出其中尚未达到最优的部分，然后对其重新进行组合，目的是改善程序的执行性能。gcc 提供的代码优化功能非常强大，它通过编译选项-On 来控制优化代码的生成，其中 n 是一个代表优化级别的整数。对于不同版本的 gcc 来讲，n 的取值范围及其对应的优化效果可能并不完全相同，比较典型的范围是从 0 变化到 2 或 3。

编译时使用选项-O 可以告诉 gcc 同时减小代码的长度和执行时间，其效果等价于-O1。在这一级别上能够进行的优化类型虽然取决于目标处理器，但一般都会包括线程跳转(Thread Jump)和延迟退栈(Deferred Stack Pops)两种优化。选项-O2 告诉 gcc 除了完成所有-O1 级别的优化之外，同时还要进行一些额外的调整工作，如处理器指令调度等。选项-O3 则除了完成所有-O2 级别的优化之外，还包括循环展开和其他一些与处理器特性相关的优化工作。通常来说，数字越大优化的等级越高，同时也就意味着程序的运行速度越快。许多 Linux 程序员都喜欢使用-O2 选项，因为它在优化长度、编译时间和代码大小之间，取得了一个比较理想的平衡点。

下面通过具体实例来感受一下 gcc 的代码优化功能，代码如程序 4.4 所示。此段程序代码故意写得效率很低，从而使读者能够在程序运行的过程中看出 gcc 优化器对程序的执行速度所起的巨大作用。读者请注意比较没有优化和经过优化的结果之间存在的显著区别。

【程序 4.4】一段效率很低的代码：inefficient_code.c。

```
#include <stdio.h>
int main(void)
{
    unsigned long int counter;                    /*定义相关的变量*/
    unsigned long int result;
    unsigned long int temp;
unsigned int five;
int i;
/*判断条件在每一次 for 循环时都会进行一次计算*/
    for (counter=0; counter < 2009 * 2009 * 100 / 4 + 2010; counter += (10 - 6) / 4)
{
            temp=counter / 1979;
for(i=0; i < 20; i++)
  {
        five=200*200/8000;          /*每一次 for 循环都会进行复杂的计算*/
}
            result=counter;
        }
    printf("Result is %ld\n", result);
    return 0;
}
```

首先不加任何优化选项进行编译：

```
# gcc -Wall inefficient_code.c -o inefficient_code
```

借助 Linux 提供的 time 命令，可以大致统计出该程序在运行时所需要的时间，注意此时最好退出其他程序，输入命令如下所示：

```
# time ./inefficient_code
```

得到如下信息：

```
Result is 100904034
real    0m8.248s
user    0m7.701s
sys     0m0.000s
```

当然，在配置不同的机器上运行，可能得到不同的时间统计结果。

说 明

Linux 的 time 命令用于测量指定程序的执行时间，结果由 3 部分组成。

- real：进程总的执行时间，它和系统负载有关(包括了进程调度，切换的时间);
- user：被测量的进程中用户指令的执行时间;
- sys：被测量进程中内核代用户指令执行的时间，user 和 sys 的和被称为 CPU 时间。

通过执行 time 命令，上面的输出内容说明了一些信息：这个程序的运行共消耗了 8 秒多的时间，其中大概有 7.7 秒的时间是用于 CPU 的运行。最后的 sys 值表示系统调用消耗的时间，说明运行这个程序时几乎所有的时间都花费在计算上面，唯一的输出发生在 printf()函数中。

现在来看看使用了优化选项后的情况。首先使用-O 选项来优化程序，命令如下：

```
# gcc -Wall -O inefficient_code.c -o inefficient_code
```

在同样的条件下再次测试一下运行时间：

```
# time ./inefficient_code
```

得到如下输出结果：

```
Result is 100904034
real    0m2.121s
user    0m2.006s
sys     0m0.000s
```

对比两次执行的输出结果不难看出，程序的性能的确得到了很大幅度的改善，由原来的 8 秒缩短为 2 秒。

步骤同上，再分别使用-O2 级别和-O3 级别对程序进行优化，并使用 time 命令查看程序的运行时间。先使用-O2 优化级别：

```
# gcc -Wall –O2 inefficient_code.c -o inefficient_code
# time ./inefficient_code
得到如下输出：
Result is 100904034
real    0m1.412s
```

```
user    0m1.302s
sys    0m0.000s
```

再使用-O3 优化：

```
# gcc -Wall –O3 inefficient_code.c -o inefficient_code
# time ./inefficient_code
```

又会得到如下输出：

```
Result is 100904034
real    0m1.400s
user    0m1.283s
sys     0m0.000s
```

从以上结果看出，使用-O2 优化级别时，程序运行所消耗的时间与-O1 优化级别相比有了很明显的改善，而-O3 优化比-O2 优化后的性能又有所提高，虽然不明显，但对于一个要消耗几分钟或者更长时间的程序，将节省大量的时间。

程序 4.4 是专门针对 gcc 的优化功能而设计的例子，因此优化前后程序的执行速度发生了很大的改变。当然在实际的程序设计过程中，程序员显然不会写出像程序 4.4 这样糟糕的代码。从程序 4.4 中可以看出存在以下问题：

- for 循环的条件判断值在每一次循环时都要进行计算，如果改变代码，直接写出该值，则代码的性能将会大大提高。当使用了优化选项后，优化器会很简单地计算出这个值，避免重复计算。
- counter 变量的步长增加值也有在每次循环中重复计算的问题。
- temp 变量对于程序的输出结果根本没有使用，但是程序在每一次循环中都会为它分配内存资源，降低了性能。
- 内嵌的 for 循环仅仅是在每次循环中重复做 20 次“five=200*200/8000”运算和赋值，而这与程序想要得到的输出结果是毫无关系的。
- 观察程序的循环体，结合以上的分析，for 循环是可以删掉的。

下面给出程序 4.4 经改动后的代码。

【程序 4.5】修改后效率较高的代码：efficient_code.c。

```
#include <stdio.h>
int main(void)
{
    long int counter = 100904034;        /*变量定义时直接给出结果，不再定义无用的变量*/
    long int result;
    result = counter;
    printf("Result is %lf\n", result);
    return 0;
}
```

进行编译：

```
# gcc -Wall efficient_code.c -o efficient_code
```

注意编译时并没有对程序进行优化。还是使用 time 命令查看程序运行所消耗的时间：

```
# time ./efficient_code
```

得到如下输出：

```
Result is 405426346
real    0m0.004s
user    0m0.000s
sys     0m0.000s
```

读者会发现程序的输出结果没有变化，但运行时间却发生了巨大的变化，即使是原来的程序经过了-O3 级别优化后的性能和现在比起来，也是望尘莫及的。

因此，尽管 gcc 的代码优化功能非常强大，但作为一名优秀的 Linux 程序员，首先还是要力求能够手工编写出高质量的代码。如果编写的代码简短，并且逻辑性强，编译器就不会做更多的工作，甚至根本不用优化。

最后，值得一提的是，优化虽然能够给程序带来更好的执行性能，但在如下一些场合中应该避免优化代码：

- 程序开发的时候。优化等级越高，消耗在编译上的时间就越长，因此在开发的时候最好不要使用优化选项，只有到软件发行或开发结束的时候，才考虑对最终生成的代码进行优化。
- 资源受限的时候。一些优化选项会增加可执行代码的体积，如果程序在运行时能够申请到的内存资源非常紧张(如一些实时嵌入式设备)，那就不要对代码进行优化，因为由此带来的负面影响可能会产生非常严重的后果。
- 跟踪调试的时候。在对代码进行优化的时候，某些代码可能会被删除或改写，或者为了取得更佳的性能而进行重组，从而使跟踪和调试变得异常困难。

4.2.5 链接库

在 Linux 下开发软件时，完全不使用第三方函数库的情况是比较少见的，通常情况下都需要借助一个或多个函数库的支持才能够完成相应的功能。从程序员的角度来看，函数库实际上就是一些头文件(.h)和库文件(.so 或者.a)的集合。

虽然 Linux 下大多数函数的头文件的默认路径是/usr/include/，而库文件的默认路径是/usr/lib/，但并不是所有的情况都是这样的。正因为如此，程序员在使用 gcc 编译时，必须为其指定所需要的头文件和库文件的路径。

gcc 采用搜索目录的办法来查找所需要的文件，-I 选项可以向 gcc 的头文件搜索路径中添加新的目录。假如在当前目录下已成功编写了 C 程序文件 foo.c，而在/home/zhangfan/include/目录下有编译该程序时所需要的头文件(不是在/usr/include/目录下)，为了让 gcc 能够顺利地找到它们，就可以使用-I 选项：

```
# gcc foo.c -I /home/zhangfan/include -o foo
```

同样，如果使用了不在标准位置的库文件，那么可以通过使用-L 选项向 gcc 的库文件搜索路径中添加新的目录。如果在/home/zhangfan/lib/目录下有 foo.c 程序链接时所需要的库文件

libfoo.so，为了让 gcc 能够顺利地找到它，并成功编译成可执行文件 foo，可以使用下面的命令：

```
# gcc foo.c -L /home/zhangfan/lib –lfoo -o foo
```

在上面的编译命令中，需要向读者说明的是-l 选项，它指示 gcc 去链接库文件 libfoo.so。

Linux 下的库文件在命名时有一个约定，那就是应该以“lib”3 个字母开头，由于所有的库文件都遵循了同样的规范，因此在用-l 选项指定链接的库文件名时可以省去“lib”3 个字母，也就是说 gcc 在对“-lfoo”字串进行处理时，会自动去链接名为 libfoo.so 的文件。

另外，Linux 下的库文件分为两大类，分别是动态链接库(通常以.so 结尾)和静态链接库(通常以.a 结尾)，两者的差别仅在于程序执行时所需的代码是在运行时动态加载的，还是在编译时静态加载的。默认情况下，gcc 在链接时优先使用动态链接库，只有当动态链接库不存在时才考虑使用静态链接库，如果需要的话可以在编译时加上-static 选项，强制使用静态链接库。例如，如果在/home/zhangfan/lib/目录下有链接时所需要的库文件 libfoo.so 和 libfoo.a，为了让 gcc 在链接时只用到静态链接库，而不使用动态链接库，可以使用下面的编译命令：

```
# gcc foo.c -L /home/zhangfan/lib -static -lfoo -o foo
```

4.2.6 同时编译多个源程序

在采用模块化的设计思想进行软件开发时，通常整个程序是由多个源文件组成的，也就相应地形成了多个编译单元，使用 gcc 能够很好地管理这些编译单元。假设有一个由 foo1.c，foo2.c，foo3.c 3 个源文件组成的程序，为了对它们进行编译，并最终生成可执行程序 foo，可以使用下面这条命令：

```
# gcc foo1.c foo2.c foo3.c -o foo
```

如果同时处理的文件不止一个，gcc 仍然会按照预处理、编译和链接的过程依次进行，上面这条命令大致相当于依次执行如下 4 条命令：

```
# gcc -c foo1.c -o foo1.o
# gcc -c foo2.c -o foo2.o
# gcc -c foo3.c -o foo3.o
# gcc foo1.o foo2.o foo3.o -o foo
```

在编译一个包含许多源文件的工程时，若只用一条 gcc 命令来完成编译是非常浪费时间的。假设项目中有 100 个源文件需要编译，并且每个源文件中都包含 10 000 行代码，如果像上面那样仅用一条 gcc 命令来完成编译工作，那么 gcc 需要将每个源文件都重新编译一遍，然后再全部链接起来。很显然，这样浪费的时间相当多，尤其是当用户只是修改了其中某一个文件的时候，完全没有必要将每个文件都重新编译一遍，因为很多已经生成的目标文件是不会改变的。要解决这个问题，关键是要灵活运用 gcc，同时还要借助像 make 这样的工具。关于 make，将在第 5 章向读者进行详细讲解。

4.2.7 管道

编译器在将源代码编译成可执行文件的过程中，需要经过许多中间步骤，包括预处理、编

译、汇编和链接。这些过程实际上是由不同的程序负责完成的。大多数情况下 gcc 可以为 Linux 程序员完成所有的后台工作，自动调用相应程序进行处理。

这样做有一个很明显的缺点，就是 gcc 在处理每一个源文件时，最终都需要生成好几个临时文件才能完成相应的工作，从而无形中增加了系统资源的开销，导致处理速度变慢。例如，gcc 在处理一个源文件时，可能需要一个临时文件来保存预处理的输出、一个临时文件来保存编译器的输出、一个临时文件来保存汇编器的输出，而读写这些临时文件显然需要耗费一定的时间。当软件项目变得非常庞大的时候，在这上面花费的代价可能会变得很沉重。

解决的办法是使用 Linux 提供的一种更加高效的通信方式——管道(pipe)。管道的实质是进程间的通信方式(读者可参见第 10 章的内容)，在这里简单地说，它可以用来同时连接两个程序(进程)，其中一个程序的输出将被直接作为另一个程序的输入，这样就可以避免使用临时文件，但编译时却需要消耗更多的内存。

在编译过程中使用管道是由 gcc 的-pipe 选项决定的。下面的这条命令就是借助 gcc 的管道功能来提高编译速度的：

```
# gcc -pipe –Wall foo.c -o foo
```

在编译小型工程时使用管道，编译时间上的差异可能还不是很明显，但在源代码非常多的大型工程中，差异将变得非常明显。

4.2.8 调试选项

在默认情况下，gcc 在编译时不会将调试符号插入到生成的二进制代码中，因为这样会增加可执行文件的大小。如果需要在编译时生成调试符号信息，可以使用 gcc 的-g 或者-ggdb 选项。

gcc 在产生调试符号时，同样采用了分级的思路，开发人员可以通过在-g 选项后附加数字 1、2 或 3 来指定在代码中加入调试信息的多少。默认的级别是 2(-g2)，此时产生的调试信息包括扩展的符号表、行号、局部或外部变量信息。级别 3(-g3)包含级别 2 中的所有调试信息，以及源代码中定义的宏。级别 1(-g1)不包含局部变量和与行号有关的调试信息，因此只能够用于回溯跟踪和堆栈转储。回溯跟踪指的是监视程序在运行过程中的函数调用历史，堆栈转储则是一种以原始的十六进制格式保存程序执行环境的方法，两者都是经常用到的调试手段。

gcc 产生的调试符号具有普遍的适应性，可以被许多调试器加以利用，但如果使用的是 gdb 调试器，那么还可以通过-ggdb 选项在生成的二进制代码中包含 gdb 专用的调试信息。这种做法的优点是可以方便 gdb 的调试工作，但缺点是可能导致其他调试器(如 DBX)无法进行正常的调试。选项-ggdb 能够接受的调试级别和-g 是完全一样的，它们对输出的调试符号有相同的影响。

需要注意的是，使用任何一个调试选项都会使最终生成的二进制文件的大小急剧增加，同时增加程序在执行时的开销，因此调试选项通常仅在软件的开发和调试阶段使用。这里还是借用程序 4.4 所示的那段糟糕的代码(inefficient_code.c)来向读者说明调试选项对生成代码大小的影响，下面编译命令，在经过未加调试选项的 gcc 编译后查看文件的大小信息：

```
# gcc -o inefficient_code inefficient_code.c
# ls -l inefficient_code
```

```
-rwxr-xr-x    1 root    root        11633   9 月   1 10:34 inefficient_code
```

在经过加入调试选项-g 的 gcc 编译后查看文件的大小信息：

```
#gcc -g -o inefficient_code inefficient_code.c
# ls -l inefficient_code
-rwxr-xr-x    1 root    root        15897   9 月   1 10:36 inefficient_code
```

从中可以看出，在未加调试选项时，生成的可执行文件 inefficient_code 的大小为 11633Byte(11.4KB)，加入调试选项-g 之后，inefficient_code 的大小变为 15897Byte(15.5KB)。可见，调试选项增加了可执行文件的大小。

虽然如此，但事实上 Linux 中的许多软件在测试版本甚至最终发行版本中仍然使用了调试选项来进行编译，这样做的目的是鼓励用户在发现问题时自己动手解决，这是 Linux 的一个显著特色。

4.3 gdb 调试器

gdb 是 Linux 下一款功能强大的 C/C++程序调试工具，本节将通过实例代码向读者介绍 gdb 的各项功能。通过本节的介绍，读者应该掌握 gdb 下的常用命令，并学会如何使用 gdb 来调试自己的源程序。

4.3.1 gdb 简介

在程序开发的过程中，调试是不可避免的步骤之一，Linux 下的 GDB(GNU Debugger)是一个用来调试 C、C++程序的功能强大的调试器，它能够在程序运行的过程中观察程序的内部结构和内存的使用情况。程序员也可以使用 gdb 来跟踪程序中的错误，从而减少程序员的工作量。

一般来说，gdb 主要提供以下功能：

- 设置断点(断点可以是条件表达式)，使程序在指定的代码行上暂停执行，便于观察。
- 单步执行程序，便于调试。
- 查看程序中变量值的变化。
- 动态改变程序的执行环境。
- 分析崩溃程序产生的 core 文件。

gdb 是一个命令行方式的调试工具，它不同于我们在 Windows 下常见的 Turbo C，VC 等图形化程序开发工具。对于 Linux 程序员来讲，gdb 通过与 gcc 的配合使用，为基于 Linux 的软件开发提供了一个完善的调试环境。

gdb 的使用非常简单，只要在 Linux 的命令提示符下输入 gdb，系统便会启动 gdb，并打印出 gdb 的相关信息：

```
#gdb
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
```

```
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.   Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu".
(gdb)
```

说 明

本书中的示例代码及相关的系统信息是在 Ubuntu 12.04LTS 版本的操作系统环境下运行产生的，可能会与其他版本的 Linux 有所不同，当然，读者不必担心这些问题，因为书中所介绍的原理在所有的 Linux 操作系统中是具有普遍性的。

也可以在 gdb 后面给出文件名，直接指定想要调试的程序，gdb 就会自动调用这个可执行文件进行调试。命令形式如下：

```
#gdb filename
```

告诉 gdb 装入名为 filename 的可执行文件进行调试。

另外，在 4.2.8 节已经提到，为了使 gdb 正常工作，必须使程序在编译的时候包含调试信息，这需要在 gcc 编译时加上-g 或者-ggdb 选项。调试信息包含了程序中的每个变量的类型和在可执行文件中的地址映射及源代码的行号。而 gdb 正是利用这些信息使源代码和机器码相关联。

4.3.2 gdb 常用命令

gdb 支持很多的命令，使用户能实现不同的功能，有简单的文件装入命令，有允许程序员检查所调用的堆栈内容的复杂命令，为方便本章后续内容的讲解和方便读者查阅，这里先将 gdb 的常用命令列出，如表 4.3 所示。想了解 gdb 的详细使用的读者还可以参考 gdb 的指南页。

表 4.3 gdb 常用命令

命　令	含 义 描 述
File	装入想要调试的可执行文件
run	执行当前被调试的程序
kill	终止正在调试的程序
step	执行一行源代码而且进入函数内部
next	执行一行源代码但不进入函数内部
break	在代码里设置断点，这将使程序执行到这里时被挂起
print	打印表达式或变量的值，或打印内存中某个变量开始的一段连续区域的值，还以用来对变量进行赋值
display	设置自动显示的表达式或变量，当程序停住或在单步跟踪时，这些变量会自动显示其当前值
list	列出产生执行文件的源代码的一部分
quit	退出 gdb

(续表)

命　　令	含 义 描 述
watch	使你能监视一个变量的值而不管它何时被改变
backtrace	回溯跟踪
frame n	定位到发生错误的代码段，n 为 backtrace 命令的输出结果中的行号
examine	查看内存地址中的值
jump	使程序跳转执行
signal	产生信号量
return	强制函数返回
call	强制调用函数
make	使用户不退出 gdb 就可以重新产生可执行文件
shell	使用户不离开 gdb 就执行 Linux 的 Shell 命令

技 巧

用户可以通过在 gdb 下输入“help”命令来查看如何使用 gdb，或者是在命令提示符下输入“gdb h”来得到一个关于 gdb 命令选项说明的简单列表。

通常情况下，gdb 的命令都可以采用简写的形式来方便用户操作。比如 run 可以简写为 r，next 可以简写为 n，backtrace 可以简写为 bt 等，读者将会在本章后续的内容中体会到这一点。

另外，gdb 支持很多与 UNIX Shell 程序一样的命令编辑特征，例如用户能够像在 bash 或 tcsh 里那样使用“Tab”键让 gdb 补齐一个唯一的命令，如果不唯一的话 gdb 会列出所有匹配的命令，用户还可以使用光标键上下翻动历史命令。

4.3.3 gdb 调试初步

本小节将通过一个具体的简单实例向读者介绍如何使用 gdb 调试器来分析程序中的错误，帮助读者快速入门。代码清单如程序 4.6 所示。

【程序 4.6】一个简单的例子认识 gdb 调试：simple_gdb.c。

```
#include <stdio.h>
int main(void)
{
    int input = 0;
    printf("Input an integer:");
    scanf("%d", input);                    /*这里出现了错误*/
    printf("The integer you input is %d\n", input);
    return 0;
}
```

使用 gcc 编译上述代码，并加上-ggdb3 调试选项，命令如下：

```
# gcc -ggdb3 simple_gdb.c -o simple_gdb
```

运行生成的可执行文件 simple_gdb，发现程序产生了一个严重的分段错误(Segmentation fault)，信息如下：

```
# ./simple_gdb
Input an integer:2009
Segmentation fault
```

为了更快速地发现错误所在，现在使用 gdb 进行跟踪调试，命令如下：

```
# gdb simple_gdb
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
……
(gdb)
```

当提示符(gdb)出现的时候，表明调试器已经做好准备进行调试了，现在可以通过 run 命令让程序开始在 gdb 的监控下运行：

```
(gdb) run
Starting program: /home/zhangfan/code/simple_gdb
Input an integer:2009
Program received signal SIGSEGV, Segmentation fault.
0x4205637f in _IO_vfscanf_internal () from /lib/tls/libc.so.6
```

仔细分析一下 gdb 给出的输出结果，不难看出，程序是由于段错误而导致异常中止的，说明内存操作出了问题，具体发生问题的地方是在调用_IO_vfscanf_internal()的时候。为了得到更加有价值的信息，可以使用 gdb 提供的回溯跟踪命令 backtrace，执行结果如下：

```
(gdb) backtrace
#0   0x4205637f in _IO_vfscanf_internal () from /lib/tls/libc.so.6
#1   0xbfffe950 in ?? ()
#2   0x42015574 in scanf () from /lib/tls/libc.so.6
#3   0x08048393 in main () at simple_gdb.c:6
#4   0x42015574 in __libc_start_main () from /lib/tls/libc.so.6
```

跳过输出结果中的前面 3 行，从输出结果的第 4 行中不难看出，gdb 已经将错误定位到 simple_gdb.c 中的第 6 行了。现在仔细检查一下：

```
(gdb) frame 3
#3   00x08048393 in main () at simple_gdb.c:6
6          scanf("%d", input);
```

使用 gdb 提供的 frame 命令可以定位到发生错误的代码段，该命令后面跟着的数值可以在 backtrace 命令输出结果中的行首找到。现在已经发现错误所在了，很明显，应该将

```
scanf("%d", input);
```

改为：

```
scanf("%d", &input);
```

这样就完成了程序的调试，退出 gdb：

```
(gdb) quit
```

修改源程序，并再次编译程序看运行是否正确：

```
# gcc -ggdb3 simple_gdb.c -o simple_gdb
# ./simple_gdb
Input an integer:2010
The integer you input is 2010
```

运行程序，得到了我们想要的结果。

当然，gdb 的功能远远不止如此，它还可以单步跟踪程序、检查内存变量、查看栈信息和设置断点等，下一节将向读者介绍 gdb 调试器的强大功能。

4.4 gdb 的使用详解

通过使用 gdb 逐步调试代码，可以看到程序内部是如何运行的，还可以查看程序中变量的值、内存使用情况、栈信息及其他一些细节问题。

下面通过一个实例来介绍 gdb 调试的具体步骤，读者通过此程序可以学到怎样跟踪程序代码，并掌握如何运用一些技巧来调试程序。源代码如程序 4.7 所示。

【程序 4.7】计算两个整数的平方和：square_sum.c。

```
#include <stdio.h>
#include <stdlib.h>
int calculate(int x,int y);
int main(void)
{
int num_1,num_2,result;
while(1)                    /*使用死循环，使程序可以一直接收终端的输入*/
{
        printf("Enter two integers,or use 0 0 to exit:");
        scanf("%d %d",&num_1,&num_2);       /*输入两个整数*/
        if (num_1==0 && num_2==0)           /*两个整数均为 0 时退出*/
          exit(0);
        result=calculate(num_1,num_2);      /*调用 calculate 函数进行计算 */
        printf("The result is:%d\n",result);    /*输出结果*/
    }s
  return 0;
}
int calculate(int x,int y)
{
 int res;
    res=x*x + y*y;
    return res;
}
```

这段代码实现的功能是，输入任意的两个整数，求它们的平方和，当两个数均为 0 时，退

出运算。阅读代码可以看到，程序使用了一个 while(1)死循环，使其可以一直接收终端的输入，并进行运算后产生输出。运算部分调用子函数 calculate()，并通过参数的值传递方式将输入的两个整数传递给它。

编译并运行程序如下：

```
# gcc -ggdb3 -o square_sum square_sum.c
# ./square_sum
Enter two integers,or use 0 0 to exit:1 2
The result is:5
Enter two integers,or use 0 0 to exit:10 20
The result is:500
Enter two integers,or use 0 0 to exit:-99 -2009
The result is:4045882
Enter two integers,or use 0 0 to exit:0 0
#
```

接下来使用 gdb 调试器看看程序内部的情况。

4.4.1 调用 gdb

要调试程序，首先要做的当然是调用调试器 gdb，装载想要调试的程序，前面已经向读者介绍，这里不再赘述。

使用命令：

```
# gdb square_sum
```

进入 gdb，系统打印信息：

```
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
……
(gdb)
```

4.4.2 使用断点

断点指出了 gdb 将要在该点处中断程序的运行，从而便于程序员单步跟踪代码。可以通过使用 break 命令，指定一个特定的位置设置断点，当程序运行到断点处时就暂停程序，然后把程序的控制权交给调试器和程序员。

用 break 命令设置断点有多种方法，表 4.4 列出了这些命令的含义。

表 4.4 break 命令的用法

命 令	含 义 描 述
break <function>	在进入指定函数时停住。C++中可以使用 class::function 或 function(type,type)格式来指定函数名
break <linenum>	在指定行号停住
break +offset	在当前行号的前面的 offset 行停住。offset 为自然数
break -offset	在当前行号的后面的 offset 行停住

(续表)

命　　令	含 义 描 述
break filename:linenum	在源文件 filename 的 linenum 行处停住
break filename:function	在源文件 filename 的 function 函数的入口处停住
break *address	在程序运行的内存地址处停住
break	该命令没有参数时，表示在下一条指令处停住
break ... if <condition>	condition 表示条件，在条件成立时停住。比如在循环体中，可以设置 break if i=100，表示当 i 为 100 时停住程序

一般来说，gdb 下的常用命令(例如 break，和下面将要提到的 list、jump 等)后面都可以跟不同的参数，使命令变得更加灵活。这些参数如表 4.5 所示。

表 4.5　gdb 常用命令的参数

参　　数	含 义 描 述
<linenum>	行号
<function>	函数名
<+offset>	当前行号的正偏移量
<-offset>	当前行号的负偏移量
<filename:linenum>	某个文件的某一行
<filename:function>	某个文件的某个子函数
<*address>	程序运行时的语句在内存中的地址

现在回到程序 4.7 的调试过程，我们在 main()函数处设置断点：

```
(gdb) break main
Breakpoint 1 at 0x80483a0: file square_sum.c, line 8.
```

然后使用 run 命令(简写为 r)开始执行程序：

```
(gdb) run
Starting program: /home/zhangfan/code/square_sum
Breakpoint 1, main () at square_sum.c:8
8    printf("Enter two integers,or use 0 0 to exit:");
```

由此可见，程序在设置的断点处停止运行了，gdb 会指出所遇到的断点，然后显示将要执行的下一行程序。

接下来可以使用单步调试命令 step(简写为 s)来跟踪程序，它一次只执行程序中的一行代码。命令如下：

```
(gdb) step
9     scanf("%d %d",&num_1,&num_2);
(gdb) s
```

```
Enter two integers,or use 0 0 to exit:-99 -2009      /*从键盘输入两个整数：-99、-2009*/
10    if (num_1==0&&num_2==0)
(gdb) s
12    result=calculate(num_1,num_2);
(gdb) s
calculate (x=-99, y=-2009) at square_sum.c:19
21    return res;
(gdb) s
22    }
(gdb)s
main () at square_sum.c:13
13    printf("The result is:%d\n",result);
(gdb) s
The result is:4045882
```

单步执行程序，最后得到了正确的结果。

4.4.3 查看运行时数据

在调试程序的过程中，往往需要查看程序中某些表达式或变量的值，以判断程序运行是否正确。使用 gdb 调试时，常用到的是 print、display 命令，以及查看内存、寄存器的信息等。

1. print 命令

在调试程序时，当程序被停住时，可以使用 print 命令(简写为 p)，或是同义命令 inspect 来查看当前程序的运行数据。print 命令的格式是：

```
print <expr>
print /<f> <expr>
```

<expr>是表达式，是所调试程序的语言的表达式(gdb 可以调试多种编程语言)；<f>是输出的格式。比如，如果要把表达式按十六进制的格式输出，那么就是/x。

还是利用程序 4.7 来进行调试，这一次我们直接将断点设置在第 9 行：

```
(gdb) break 9                                    /*断点设置在第 9 行*/
Breakpoint 1 at 0x80483b0: file square_sum.c, line 9.
(gdb) r
Starting program: /home/zhangfan/code/square_sum
Breakpoint 1, main () at square_sum.c:9
9     scanf("%d %d",&num_1,&num_2);
(gdb) s
Enter two integers,or use 0 0 to exit:-99 -2009    /*从键盘输入两个整数：-99、-2009*/
10    if (num_1==0&&num_2==0)
(gdb) print num_1                                /*打印变量 num_1 的值*/
$1=-99
(gdb) print num_2                                /*打印变量 num_2 的值*/
$2=-2009
```

从以上代码的最后两行输出可以看到，当从键盘输入两个整数-99 和-2009 分别赋值给变量 num_1 和 num_2 后(由程序完成)，再使用命令“print num_1” 和“print num_2”打印出它们的值，得到了正确的结果。

当使用 gdb 的 print 命令查看程序运行时的数据时，每一个 print 都会被 gdb 记录下来。gdb 会以$1, $2, $3…这样的方式为每一个 print 命令编上号。于是，可以使用这个编号访问以前的表达式，如$1。例如：

```
(gdb) print $1                  /*$1 代表了变量 num_1 的值*/
$1 = -99
(gdb) print $2                  /*$2 代表了变量 num_2 的值*/
$2 = -2009
```

这个功能所带来的好处是，如果先前输入了一个比较长的表达式，并想查看这个表达式的值，可以使用历史记录来访问，避免了重复输入。

另外，要注意 print 命令的表达式中两个具有特殊意义的符号，它们是$和$$。$表示给定序号的前一个序号，而$$表示给定序号的向前两个的序号，如果不给定序号，gdb 将默认当前序号为给定序号。还是看 num_1 的例子：

```
(gdb) print num_1                  /*打印变量 num_1 的值*/
$1 = -99
(gdb) print $
$2 = -99
(gdb) print $$
$3 = -99
(gdb) print $$3
$4= -99
```

可以看出，输入“print $”时的当前序号为 2，而$表示 2 的前一个序号，即为 1；输入“print $$”时的当前序号为 3，而$$表示给定序号的向前两个的那个序号，这里即为 1；输入“print $$3”时，给定了序号 3，这样$$3 就表示了序号 1。由此，上面的 4 条输出均为-99。

print 命令是 gdb 下最常用的命令之一，它的功能除了打印表达式或变量的值外，还有对变量进行赋值和打印内存中某个变量开始的一段区域的内容。这将在本小节后面的内容中向读者提到。

2. 输出格式

上一小节中已向大家提到了 print 命令的输出格式。一般来说，gdb 的命令会根据变量的类型输出变量的值，但也可以自定义 gdb 的输出格式。例如，想输出一个整数的十六进制或是二进制来查看这个整型变量中的位的情况。要做到这样，可以使用 gdb 的数据显示格式，如表 4.6 所示。

表 4.6　gdb 的数据显示格式

符　　号	含　　义
x	按十六进制格式显示变量
d	按十进制格式显示变量
u	按十六进制格式显示无符号整型
o	按八进制格式显示变量
t	按二进制格式显示变量

(续表)

符　号	含　义
a	按十六进制格式显示变量
c	按字符格式显示变量
f	按浮点数格式显示变量

还是看看程序 4.7 中的变量 num_1 在不同的输出格式下打印出的值。注意此时从键盘输入给 num_1 的值已变为 55，读者可自行运算来检验显示是否正确。

```
(gdb) break 9                          /*断点设置在第 9 行*/
Breakpoint 1 at 0x80483b0: file square_sum.c, line 9.
(gdb) r
Starting program: /home/zhangfan/code/square_sum
Breakpoint 1, main () at square_sum.c:9
9       scanf("%d %d",&num_1,&num_2);
(gdb) s
Enter two integers,or use 0 0 to exit:55 66     /*从键盘输入两个整数：55、66*/
10      if (num_1==0&&num_2==0)
(gdb) p num_1
$1=55
(gdb) p/a num_1
$2=0x37
(gdb) p/c num_1
$3=55 '7'
(gdb) p/f num_1
$4=7.70714155e-44
(gdb) p/x num_1
$5=0x37
(gdb) p/t num_1
$6=110111
```

3. 自动显示命令 display

可以设置一些自动显示的变量，当程序停住时，或是在单步跟踪时，这些变量会自动显示。相关的 gdb 命令是 display，格式如下：

```
display <expr>
display/<fmt> <expr>
display/<fmt> <addr>
```

expr 是一个表达式，fmt 表示显示的格式，addr 表示内存地址。当用 display 设定好了一个或多个表达式后，只要程序停下来，gdb 会自动显示所设置的这些表达式的值。

格式 i 和 s 同样被 display 支持，一个非常有用的命令是：

```
display/i $pc
```

$pc 是 gdb 的环境变量，表示指令的地址，/i 则表示输出格式为机器指令码，也就是汇编。于是当程序停下后，就会出现源代码和机器指令码相对应的情形，这是一个很有意思的功能。

表 4.7 是一些和 display 相关的 gdb 命令。

表 4.7　display 相关命令

命　　令	含 义 描 述
undisplay <dnums...> delete display <dnums...>	删除自动显示，dnums 为已设置好了的自动显示的编号。如果要同时删除几个编号，可以用空格分隔；如果要删除一个范围内的编号，可以用减号表示(如：2-5)
disable display <dnums...> enable display <dnums...>	不删除自动显示的设置，而只是让其失效或恢复
info display	查看 display 设置的自动显示的信息。gdb 会显示出一张表格，报告调试中设置了多少个自动显示设置，其中包括已设置的编号、表达式及是否 enable 等

下面是使用 display 命令的例子：

```
(gdb) break 9                                    /*断点设置在第 9 行*/
Breakpoint 1 at 0x80483b0: file square_sum.c, line 9.
(gdb) r
Starting program: /home/zhangfan/code/square_sum
Breakpoint 1, main () at square_sum.c:9
9     scanf("%d %d",&num_1,&num_2);
(gdb) s
Enter two integers,or use 0 0 to exit:-99 -2009  /*从键盘输入两个整数：-99、-2009*/
10    if (num_1==0&&num_2==0)
(gdb) display num_1                              /*将 num_1 设置成为自动显示的变量*/
1: num_1=-99
(gdb) display num_2                              /*将 num_2 设置成为自动显示的变量*/
2: num_2=-2009
(gdb) s
12        result=calculate(num_1,num_2);
2: num_2=-2009
1: num_1=-99
(gdb) s
calculate (x=-99, y=-2009) at square_sum.c:20
20        res=x*x+y*y;
(gdb) s
21        return res;
(gdb) s
22        }
(gdb) s
main () at square_sum.c:13
13      printf("The result is:%d\n",result);
2: num_2=-2009
1: num_1=-99
(gdb) s
The result is:4045882
```

从上面的代码可以看出，将 num_1、num_2 设置成为自动显示的变量后，在程序以后的单

步跟踪过程中，每一步的执行结果都会显示 num_1 和 num_2 的值，直到退出 gdb 为止。

细心的读者会发现，在程序运行的第 20～22 行并没有显示 num_1 和 num_2 这两个变量值。原因是 num_1 和 num_2 是 main()函数中定义的变量，当程序进入 calculate()(第 20 行开始)函数后，自然不会显示它们了。

4. 查看内存

可以使用 examine 命令(简写是 x)来查看内存地址中的值。x 命令的语法如下所示：

```
x/<n/f/u> <addr>
```

对于该命令的说明如下：

- n、f、u 是可选的参数，可以独立使用，也可联合使用。
- n 是一个正整数，表示显示内存的长度，也就是说从当前地址向后显示几个地址的内容。
- f 表示显示的格式，参见上面。如果地址所指的是字符串，那么格式可以是 s，如果是指令地址，那么格式可以是 i。
- u 表示从当前地址往后请求的字节数，如果不指定的话，gdb 默认是 4 个 bytes。u 参数可以用下面的字符来代替：b 表示单字节，h 表示双字节，w 表示四字节，g 表示八字节。当指定了字节长度后，gdb 会从指定的内存地址开始，读写指定字节，并把其当作一个值取出来。
- <addr>表示一个内存地址。例如，有如下命令：

```
x/4uh 0x48723
```

表示从内存地址 0x48723 读取内容，h 表示以双字节为 1 个单位，4 表示 4 个单位，u 表示按十六进制显示。

5. gdb 环境变量

可以在 gdb 的调试环境中定义自己的变量，用来保存一些调试程序中的运行数据。

要定义一个 gdb 的变量很简单，只需使用 gdb 的 set 命令。gdb 的环境变量和 UNIX 一样，也是以$起始。例如：

```
set $foo = *object_ptr
```

使用环境变量时，gdb 会在第一次使用时创建这个变量，而在以后的使用中，则直接对其赋值。环境变量没有类型，可以给环境变量定义任意的类型，包括结构体和数组。

```
show convenience
```

该命令查看当前所设置的所有的环境变量。

这是一个比较强大的功能，环境变量和程序变量的交互使用将使得程序调试更为灵活便捷。例如：

```
set $i=0
print bar[$i++]->contents
```

输入这样的命令后，只用按“Enter”键，重复执行上一条语句，环境变量会自动累加，从而完成逐个输出的功能。

6. 查看寄存器

在调试程序的过程中，有时候亦需要查看某些寄存器中的值。寄存器中放置了程序运行时的数据，比如程序当前运行的指令地址(IP)，程序的当前堆栈地址(SP)等。

可以使用 info 命令来实现。表 4.8 所示为使用 info 来查看寄存器的常用命令形式。

表 4.8 查看寄存器常用命令

命　令	含 义 描 述
info registers	查看寄存器的情况(除了浮点寄存器)
info all-registers	查看所有寄存器的情况(包括浮点寄存器)
info registers <regname ...>	查看所指定的寄存器的情况，regname 表示寄存器名，多个寄存器之间用逗号隔开

同样可以使用 print 命令来访问寄存器的情况，只需要在寄存器名字前加一个$符号就可以了，例如：

```
print $ip
```

4.4.4 查看源程序

在程序调试的过程中，有时候需要查看源程序的内容，以及源代码在内存中的情况，本小节向读者介绍与此相关的 gdb 命令。

1. 显示源代码

gdb 可以打印出所调试程序的源代码，当然，在程序编译时一定要加上-g 参数，把源程序信息编译到执行文件中，不然就看不到源程序了。当程序停下来以后，gdb 会报告程序停在了程序的第几行上。可以用 list 命令来显示程序的源代码，如表 4.9 所示。

表 4.9 list 命令

命　令	含 义 描 述
list <linenum>	显示程序第 linenum 行的周围的源程序
list <function>	显示函数名为 function 的函数的源程序
list	显示当前行后面的源程序
list -	显示当前行前面的源程序。一般是显示当前行的上 5 行和下 5 行，或者显示当前行的上 2 行和下 8 行，默认共显示 10 行
set listsize <count>	设置一次显示源代码的行数
show listsize	查看当前 listsize 的设置
list <first>, <last>	显示从 first 行到 last 行之间的源代码

（续表）

命　　令	含 义 描 述
list , <last>	显示从当前行到 last 行之间的源代码
list +	向后显示源代码

还是以程序 4.7 为例来向大家介绍 list 命令的用法和功能：

```
(gdb) list
1       #include <stdio.h>
2       int calculate(int x,int y);
3       int main(void)
4       {
5           int num_1,num_2,result;
6           while(1)
7               {
8                   printf("Enter two integers,or use 0 0 to exit:");
9                   scanf("%d %d",&num_1,&num_2);
10                  if (num_1==0&&num_2==0)
(gdb) show listsize                         /*查看当前 listsize 的设置*/
Number of source lines gdb will list by default is 10.
(gdb) set listsize 5                        /*将当前 listsize 的大小设置为 5*/
(gdb) list
11                      exit(0);
12                  result=calculate(num_1,num_2);
13                  printf("The result is:%d\n",result);
14              }
15                  return 0;
(gdb) list calculate                /*只查看函数 calculate()的源代码，并且一次只显示 5 行*/
17      int calculate(int x,int y)
18      {
19          int res;
20          res=x*x+y*y;
21          return res;
```

2. 源代码的内存

可以使用 info line 命令来查看源代码在内存中的地址。和大多数 gdb 命令相同，info line 后面也可以跟“行号”、“函数名”、“文件名：行号”、“文件名：函数名”的参数形式，这个命令会显示出所指定的源代码在运行时的内存地址，来看一下程序 4.7 中的子函数 calculate()在内存中的地址：

```
(gdb) info line square_sum.c:calculate
Line 18 of "square_sum.c" starts at address 0x8048407 <calculate>
    and ends at 0x804840d <calculate+6>.
```

从上面的输出信息可以看出，calculate()在内存中的起始地址为 0x8048407，终止地址为 0x804840d。

还有一个命令 disassemble，可以查看源程序的当前执行时的机器码，这个命令会把目前内存中的指令 dump 出来。如下面的示例表示查看函数 calculate()的汇编代码：

```
(gdb) disassemble calculate
Dump of assembler code for function calculate:
0x08048407 <calculate+0>:     push    %ebp
0x08048408 <calculate+1>:     mov     %esp,%ebp
0x0804840a <calculate+3>:     sub     $0x4,%esp
0x0804840d <calculate+6>:     mov     0x8(%ebp),%eax
0x08048410 <calculate+9>:     mov     %eax,%edx
0x08048412 <calculate+11>:    imul    0x8(%ebp),%edx
0x08048416 <calculate+15>:    mov     0xc(%ebp),%eax
0x08048419 <calculate+18>:    imul    0xc(%ebp),%eax
0x0804841d <calculate+22>:    lea     (%eax,%edx,1),%eax
0x08048420 <calculate+25>:    mov     %eax,0xfffffffc(%ebp)
0x08048423 <calculate+28>:    mov     0xfffffffc(%ebp),%eax
0x08048426 <calculate+31>:    leave
0x08048427 <calculate+32>:    ret
End of assembler dump.
(gdb)
```

disassemble 是一个非常有用的命令，对于那些熟悉汇编的 C 程序员，他们在编写大型程序的时候，往往要考虑系统硬件内存资源的分配等问题，这时候利用反汇编来查看当前程序执行的机器码，对于程序的检错纠错是很有帮助的。

4.4.5 改变程序的执行

一旦使用 gdb 挂上被调试程序，当程序运行起来后，可以根据自己的调试思路来动态地在 gdb 中更改当前被调试程序的运行线路或是其变量的值。这个强大的功能能够让用户更好地调试程序。例如，可以在程序的一次运行中走遍程序的所有分支。

1. 修改变量值

在 4.4.3 小节中已向读者提到，print 命令还有一个功能是修改被调试程序中运行时的变量值，例如：

```
(gdb) print x=8
```

x=8 这个表达式是 C/C++的语法，意为把变量 x 的值修改为 8，如果当前调试的语言是 Pascal，那么可以使用 Pascal 的语法 x:=8。

下面看看 print 命令的这个功能在程序 4.7 中的使用：

```
(gdb) break 9                                    /*断点设置在第 9 行*/
Breakpoint 1 at 0x80483b0: file square_sum.c, line 9.
(gdb) r
Starting program: /home/zhangfan/code/square_sum
Breakpoint 1, main () at square_sum.c:9
9       scanf("%d %d",&num_1,&num_2);
(gdb) s
Enter two integers,or use 0 0 to exit:-99 -2009    /*从键盘输入两个整数：-99、-2009*/
```

```
10      if (num_1==0&&num_2==0)
(gdb) p num_1                    /*查看变量 num_1 的值*/
$1=-99
(gdb) p num_2                    /*查看变量 num_2 的值*/
$2=-2009
(gdb) print num_1=10             /*修改变量 num_1 的值为 10*/
$3=10
(gdb) print num_2=20             /*修改变量 num_2 的值为 20*/
$4 = 20
(gdb) s
12          result=calculate(num_1,num_2);
(gdb) s
calculate (x=10, y=20) at square_sum.c:20
20              res=x*x+y*y;
(gdb) s
21              return res;
(gdb) s
22          }
(gdb) s
main () at square_sum.c:13
13          printf("The result is:%d\n",result);
(gdb) s
The result is:500
```

从中可以看到，在调试过程中修改了变量 num_1 和 num_2 的值，程序最后的输出 500 是变量修改后的运算结果。

2. 跳转执行

一般来说，被调试程序会按照程序代码的运行顺序依次执行。gdb 提供了乱序执行的功能，也就是说，gdb 可以修改程序的执行顺序，可以让程序执行随意跳跃。这个功能可以由 gdb 的 jump 命令来实现：

```
jump <linespec>
```

指定下一条语句的运行点。<linespce>可以是文件的行号，可以是 file:line 格式，可以是+num 这种偏移量格式，表示下一条运行语句从哪里开始。

```
jump <address>
```

这里的<address>是代码行的内存地址。

注 意

jump 命令不会改变当前的程序栈中的内容，所以，从一个函数跳到另一个函数时，当函数运行完返回进行弹栈操作时必然会发生错误，可能结果还是非常奇怪的，甚至产生程序 Core Dump。所以最好是在同一个函数中进行跳转。

熟悉汇编的人都知道，程序运行时，有一个寄存器用于保存当前代码所在的内存地址。所以，jump 命令也就是改变了这个寄存器中的值。可以使用 set $pc 来更改跳转执行的地址。例如：

```
set $pc = 0x485
```

3. 产生信号量

使用 singal 命令可以产生一个信号量给被调试的程序。如中断信号 Ctrl+C。这非常方便程序的调试，可以在程序运行的任意位置设置断点，并在该断点用 gdb 产生一个信号量。精确地在某处产生信号非常有利程序的调试。其语法是：

```
signal <singal>
```

Linux 的系统信号量通常为 1～15。所以<singal>的取值也在这个范围。

signal 命令和 Shell 的 kill 命令不同，系统的 kill 命令发信号给被调试程序时，是由 gdb 截获的，而 signal 命令所发出的信号则是直接发给被调试程序的。

4. 强制函数返回

如果调试断点在某个函数中，还有语句没有执行完，可以使用 return 命令强制函数忽略还没有执行的语句并返回。

```
return
return <expression>
```

使用 return 命令取消当前函数的执行，并立即返回。如果指定了<expression>，那么该表达式的值会被当作函数的返回值。

5. 强制调用函数

强制调用函数使用 call 命令，格式如下：

```
call <expr>
```

表达式中也可以是函数，以达到强制调用函数的目的，显示函数的返回值，如果函数返回值是 void，那么就不显示。

另一个相似的命令也可以完成这一功能——print。print 后面可以跟表达式，所以也可以用它来调用函数。print 和 call 的不同之处是如果函数返回 void，call 则不显示，print 则显示函数返回值，并把该值存入历史数据中。

6. 在不同的语言中使用 gdb

gdb 支持下列语言：C、C++、Fortran、PASCAL、Java、Chill、assembly 和 Modula-2。一般来说，gdb 会根据所调试的程序来确定所用的调试语言。例如：发现文件名后缀为.c，gdb 会认为是 C 程序；文件名后缀为.C、.cc、.cp、.cpp、.cxx、.c++，gdb 会认为是 C++程序；后缀是.f、.F，gdb 会认为是 Fortran 程序；后缀为.s、.S 会认为是汇编语言。

也就是说，gdb 会根据所调试的程序的语言，来设置自己的语言环境，并让 gdb 的命令跟着语言环境的改变而改变。比如一些 gdb 命令需要用到表达式或变量时，这些表达式或变量的语法，完全是根据当前的语言环境而改变的。例如，C/C++中对指针的语法是*p，而在 Modula-2 中则是 p^。并且，如果当前的程序是由几种不同语言一同编译成的，到调试过程中，gdb 也能根据不同的语言自动地切换语言环境。这种跟着语言环境而改变的功能，确实是一种体贴开发人员的设计。

表 4.10 是几个关于 gdb 语言环境的命令。

表 4.10　gdb 的语言环境命令

命　令	含 义 描 述
show language	查看当前的语言环境。如果 gdb 不能识别所调试的编程语言，那么，C 语言被认为是默认的环境
info frame	查看当前函数的程序语言
info source	查看当前文件的程序语言

如果 gdb 没有检测出当前的程序语言，那么用户也可以手动设置当前的程序语言。使用 set language 命令即可做到。

如果 set language 命令后什么也不跟，可以查看 gdb 所支持的语言种类：

```
(gdb) set language
The currently understood settings are:

local or auto    Automatic setting based on source file
c                Use the C language
c++              Use the C++ language
asm              Use the Asm language
fortran          Use the Fortran language
java             Use the Java language
modula-2         Use the Modula-2 language
pascal           Use the Pascal language
scheme           Use the Scheme language
```

可以在 set language 后加上被列出来的程序语言名，来设置当前的语言环境。

gdb 是一个强大的命令行调试工具。大家知道命令行的强大在于其可以形成执行序列，形成脚本。Linux 下的软件以命令行的较多，这给程序开发提供了极大的便利。命令行软件的优势在于它们可以非常容易地集成在一起，使用几个简单的已有工具的命令，就可以实现一个非常强大的功能。

因此，Linux 下的软件比 Windows 下的软件更能有机地结合，各自发挥长处，组合起来具有更为强大的功能。而 Windows 下的图形软件基本上是各自为营，互相不能调用，很不利于各种软件的相互集成。这里并不是要和 Windows 进行比较，所谓“寸有所长，尺有所短”，图形化工具还是有不如命令行的地方。

4.5 xxgdb 调试器简介

在 Linux 平台下调试 C 程序时，除了使用 gdb 之外，还可以使用 xxgdb。xxgdb 是 XWindow 系统的调试工具，实际上，xxgdb 是 gdb 的图形界面版本，它保留了 gdb 的所有功能和特性。程序员可以像在 VC、Turbo C 上调试 C 程序那样，用单击按钮来代替输入命令，它还能显示当前断点设置的位置，而不用通过输入断点的名字来查询。对于已经习惯并且喜好使用图形界

面的调试工具的用户，xxgdb 将是一个不错的选择。

用户可以通过如下的命令来在线安装 xxgdb 工具。

```
sudo apt-get install xxgdb
```

xxgdb 调试器的运行界面如图 4.1 所示，要使用 xxgdb 完成调试功能，首先要对其进行初始化，用户可以使用 gdb 里任何有效的命令行选项来初始化 xxgdb。鉴于篇幅有限，对于 xxgdb 这里不进行过多的介绍，有兴趣的读者可参考其他相关书籍和资料。而本书中所有实例代码的调试都是在 gdb 调试器下进行的。

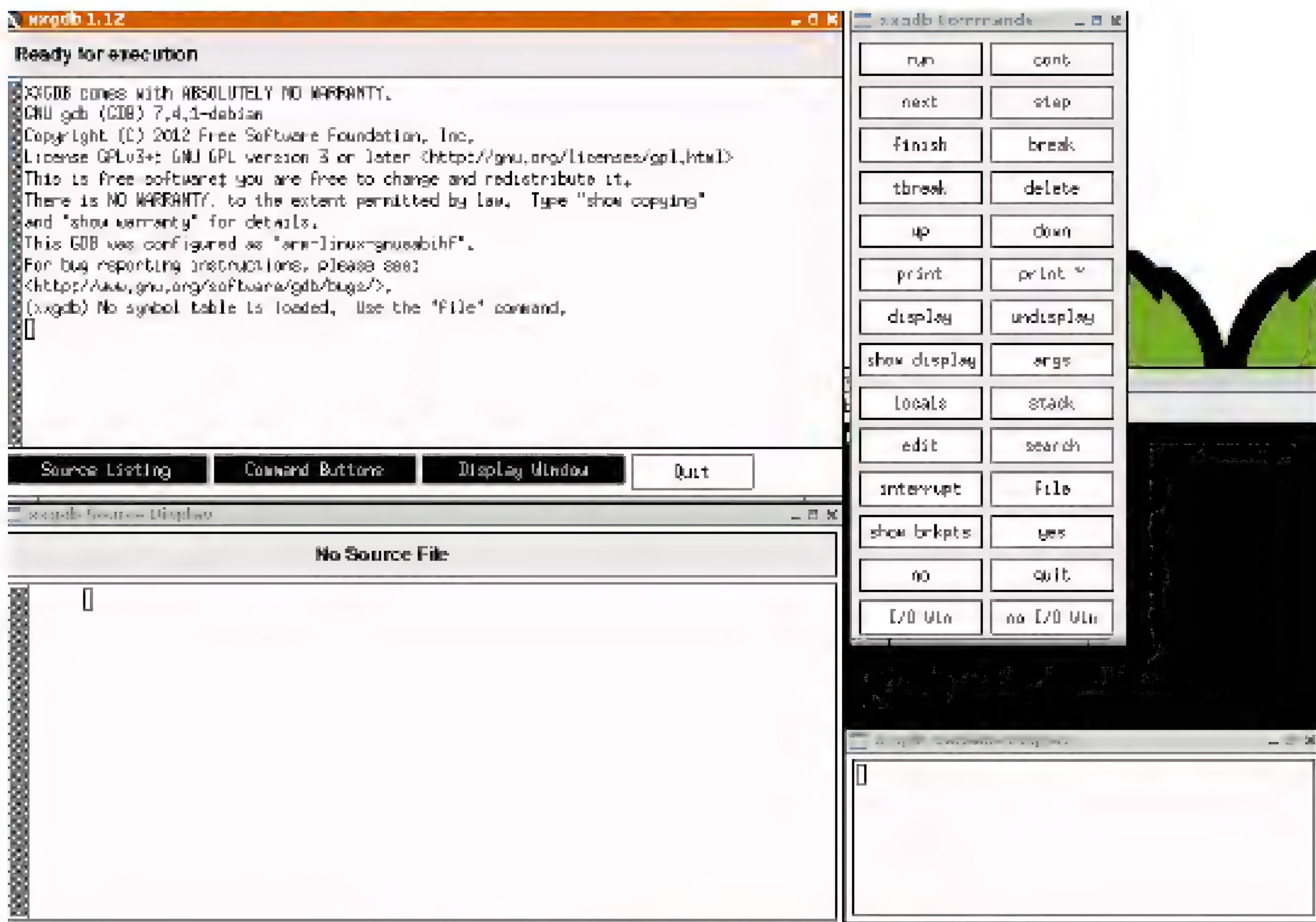


图 4.1　xxgdb 的运行界面

4.6 本章小结

本章首先从一个简单的 C 程序入手，引出 Linux 下的 C 程序标准编译器 gcc。具体介绍了 gcc 的用法，包括它的属性选项、代码优化和其他的高级应用。接着介绍了 Linux 下的 C 程序标准调试器 gdb，包括它的调试步骤，并深入地阐述了 gdb 的各项功能和用法。最后，还简单介绍了另外一种图形界面调试工具 xxgdb。

实战演练

1. 编写一个程序，使用 gcc 编译该程序，要求运行程序显示下面的结果：

```
1+1=2
2+1=3   2+2=4
3+1=4   3+2=5   3+3=6
4+1=5   4+2=6   4+3=7   4+4=8
```

2. 在下面的程序中使用了一个自己定义的头文件 myhead.h，主程序 main.c 如下：

```
#include <myhead.h>
main()
{
```

```
    int i=2010;
    double j=2010.0;
    printf("%d %f",abs(i),fabs(j));
}
```

头文件 myhead.h 的内容如下：

```
/************myhead.h************/
#include <stdio.h>
#include <math.h>
```

将这两个源代码文件存放在不同的路径下，在编译 C 程序时，试使用 gcc 的-I 选项连接头文件，使程序成功编译并运行。

3. 使用 gcc 的-pedantic 选项编译下面这段程序，注意观察编译时将会产生怎样的警告信息：

```
#include <stdio.h>
int main(void)
{
    int x=2010;
    int y=0,z;
    z=x/y;
    printf("%d",z);
}
```

改正程序后再次使用 gcc 的-pedantic 选项进行编译，直至警告全部消除。

4. 使用命令查看本机中 gcc 编译器的版本号，以及 gdb 调试器的启动信息。

5. 在下面的程序中出现了几处错误：

```
#include <stdio.h>
main()
{
    int num1,num2;
    printf("Input two numbers: ");
    scanf("%d %d",num1,num2);
    max(num1, num2)
}
int max(int i, int j)
{
    if(i>j),
        return(i);
    else
        return(j);
}
```

试使用 gdb 调试器找出这些错误，并重新编译运行改正后的程序。

6. 在第 1 题中，假设程序中加法运算求得的和存放在变量 k 中，试使用 gdb 的 print 命令查看变量 k 的值。

7. 在第 1 题中，使用 gdb 的 display 命令使变量 k 的值在调试过程中自动显示。

8. 使用 gdb 查看第 5 题改正后的源代码中 int max(int i, int j)子函数的内存情况。

第5章

make的使用和Makefile的编写

make 工程管理器是 Linux 下的一个“自动编译管理器”，“自动”是指它能够根据文件的时间戳，自动发现更新过的文件而减少程序编译的工作量。同时，它通过读入 Makefile 文件的内容来执行大量的编译工作，用户只需编写一次简单的编译语句即可。make 工具大大提高了实际项目的工作效率，几乎所有 Linux 下的项目编程都会涉及它。



本章内容：

◎ 什么是 make。
◎ Makefile 的书写规则。
◎ Makefile 中使用变量。
◎ Makefile 中的常用函数调用。
◎ Makefile 的隐式规则。

5.1 什么是 make

在第 4 章已经向读者提到，当 gcc 在编译一个包含许多源文件的工程时，gcc 需要将其中的每个源文件都编译一遍，然后再全部链接起来，这样做显然很浪费时间，尤其是当用户只是修改了其中某一个文件的时候，完全没有必要将每个文件都重新编译一遍，因为很多已经生成的目标文件是不会改变的。要解决这个问题，就要借助本章将要向读者介绍的 make 工程管理器工具。

make 和 Makefile 提供了一种非常简单有效的工程管理方式。使用这种方式管理工程的原理很简单，Makefile 是一个决定怎样编译工程的文本文件，有一定的书写规则。在工程更新的时候，使用 GNU 的 make 工具根据当前的 Makefile 对工程进行编译。

5.1.1 make 机制概述

在 Linux 的程序开发环境下，一般不具有集成开发环境(IDE)。因此，当需要大量编译工程文件的时候，就需要使用自己的方法来管理。

make 工具最初设计的目的是为了维护 C 程序文件，防止不必要的重新编译。例如，在使用命令行进行编译的时候，修改了一个工程中的头文件，如何确保包含这个头文件的所有文件都得到编译呢？这些工作可以让 make 程序来自动完成。make 工具对于维护一些具有相互依赖关系的文件特别有用，它对文件和命令的联系(在文件改变时调用来更新其他文件的程序)提供一套编码方法。在使用的过程中只告诉 make 需要做什么，即提供一些规则，其他的工作则由 make 自动完成。

make 工具的工作是自动确定工程的哪部分源程序文件需要重新编译，然后执行命令去编译它们。虽然这种方式多用于 C 程序，然而只要提供命令行的编译器，就可以将其用于任何语言。实际上，make 工具不仅应用于编程，也可以用于描述一些文件改变时，需要自动更新另一些文件的任务。在程序开发的过程中，Makefile 带来的好处就是自动化编译。当编译规则制定完成后，只需要一个 make 命令，整个工程就会根据 Makefile 判断是否需要更新来完成自动编译，极大地提高了软件开发的效率，降低了开发的复杂度。

make 机制的运行环境需要一个命令行程序 make 和一个文本文件 Makefile。

make 是一个命令工具，具体来说是一个解释 Makefile 中的指令的命令工具。Makefile 的工作原理是调用系统中的 make 命令解释当前的 Makefile，完成其中指定的功能。在很多的 IDE 中都有这个命令，如 Delphi 的 make，Visual C++的 nmake，Linux 下 GNU 的 make。可见，make 与 Makefile 已经成为一种在工程编译方面的常用方法。

提 示

make 命令执行后有 3 个退出码：

- 表示成功执行。
- 如果 make 运行时出现任何错误，则返回 1。
- 如果使用了 make 的“-q”选项，并且 make 使得一些目标不需要更新，那么返回 2。

下面通过一个简单的例子来向读者说明 make 的工作机制，如程序 5.1 所示。该程序的主要功能是由用户输入一个字符串，内有若干个字符，然后再输入一个字符，要求程序将字符串中的该字符删去，最后打印处理后的字符串。

为了演示 make 工具编译多个源程序文件的使用方法，我们将程序 5.1 分为 4 个 C 源文件，分别为 main.c、foo1.c、foo2.c 和 foo3.c。不同的 C 文件负责完成不同的功能模块：main.c 为主函数，主要是对各个程序模块的调用；foo1.c 实现字符串的输入；foo2.c 负责删除字符串中的某些特殊字符(这个字符也由用户输入)；foo3.c 输出处理后的字符串。

【程序 5.1】 make 使用实例：

```
/************************************main.c***********************************/
#include <stdio.h>
int main(void)
{
    char c;
    char str[20];
    enter_string(str);              /*调用字符串输入函数*/
    printf("The delete atring is: ");
    scanf("%c",&c);
    delete_string(str,c);           /*调用字符删除函数*/
    print_string(str);              /*打印处理后的字符串*/
    return 0;
}
/************************************foo1.c***********************************/
#include <stdio.h>
int enter_string(char str[20])
{
    printf("Input the strings: ");
    gets(str);
    return 0;
}
/************************************foo2.c***********************************/
int delete_string(char str[], char ch)
{
    int i,j;
    for(i=j=0; str[i]!='\0'; i++)
        if(str[i]!=ch)
            str[j++]=str[i];
    str[j]='\0';
    return 0;
```

```
}
/**********************************foo3.c**********************************/
#include <stdio.h>
int print_string(char str[])
{
    printf("Result: %s\n",str);
    return 0;
}
```

为了使用 make 自动编译这些源文件，我们需要书写 make 工具所需要遵循的编译规则，即 Makefile 文件。对于程序 5.1，我们可以写出下面这样的 Makefile(对于 Makefile 文件的书写规则，我们在稍后将会讲到)：

```
/**********************************Makefile**********************************/
all : main.c foo1.c foo2.c foo3.c
gcc main.c foo1.c foo2.c foo3.c -o all
```

需要提醒读者的是，这 5 个文件应当存放在 Linux 的同一个目录下，这样，当我们使用 make 进行自动编译的时候，它便会在这个目录下找到 Makefile 文件，并按照 Makefile 文件中的内容(编译规则的集合)进行自动编译。

对于程序 5.1，当我们在这 5 个文件的存放目录下输入 make 命令进行编译时，产生的输出结果如下：

```
# make
gcc main.c foo1.c foo2.c foo3.c -o all
```

从中可以看到，make 自动执行了 Makefile 中的编译命令，并生成了最终的可执行文件 all(即 Makefile 的目标——稍后会讲到 Makefile 目标的概念)。运行这个目标，得到以下输出结果：

```
# ./all
Input the strings: abcdefgabc
The delete atring is: c
Result: abdefgab
```

从中可以看到，整个程序正确地执行了。

所以，当我们在管理和编译由多个源文件组成的工程项目时，make 的自动编译和管理功能为程序员提供了一种强有力的手段。那么，make 到底是怎么工作的？make 与 Makefile 的关系又是怎样的呢？编写 Makefile 时又有什么样的规则呢？下面将向读者一一阐述这些问题。

5.1.2　make 与 Makefile 的关系

make 是一个 Linux 下的二进制程序，用来处理 Makefile 这种文本文件。在 Linux 的 Shell 命令行键入 make 的时候，将自动寻找名称为“Makefile”的文件作为编译文件，如果没有名称为“Makefile”的文件，将继续查找名称为“makefile”的文件。找到编译文件后，make 工具将根据 Makefile 中的第一个目标自动寻找依赖关系，找出这个目标所需要的其他目标。如果所需要的目标也需要依赖其他的目标，make 工具将一层层寻找直到找到最后一个目标为止。

make 工具的使用格式为：

```
make [options] [target] ...
```

options 为 make 工具的参数选项，target 为 Makefile 中指定的目标。表 5.1 给出了 make 工具的参数选项。

表 5.1 make 工具的参数选项

选　项	含　义
-f filename	显式地指定文件作为 Makefile
-C dirname	制定 make 在开始运行后的工作目录为 dirname
-e	不允许在 Makefile 中替换环境变量的赋值
-k	执行命令出错时，放弃当前目标，继续维护其他目标
-n	按实际运行时的执行顺序模拟执行命令(包括用@开头的命令)，没有实际执行效果，仅仅用于显示执行过程
-p	显示 Makefile 中所有的变量和内部规则
-r	忽略内部规则
-s	执行但不显示命令，常用来检查 Makefile 的正确性
-S	如果执行命令出错就退出
-t	修改每个目标文件的创建日期
-I	忽略运行 make 中执行命令的错误
-V	显示 make 的版本号

下面通过一个实例来讲述 make 与 Makefile 文件的关系。程序 5.2 来源于 GNU 的 make 使用手册，工程中有 8 个 C 文件和 3 个头文件，要写一个 Makefile 文件来告诉 make 命令如何编译和连接这几个文件。

说明

在程序 5.2 中，我们没有必要给出 8 个 C 文件和 3 个头文件的源代码，这里只是讲解 Makefile 文件的书写规则，以及 make 如何遵循 Makefile 的内容来进行编译工作。事实上，读者可以把这几个源代码文件想象成任何可行的代码。

在给出程序 5.2 之前，读者首先需要明白 make 操作管理 Makefile 文件的规则是：

- 如果这个工程没有编译过，那么所有 C 文件都需要被编译和连接。
- 如果这个工程中的某几个 C 文件被修改，则只需编译被修改过的 C 文件，并连接目标程序。
- 如果这个工程的头文件被改变了，则需要编译引用了这几个头文件的 C 文件，并连接目标程序。

只要 Makefile 文件写得足够好，所有的这一切只用一个 make 命令就可以完成，make 命令会自动智能地根据当前文件的修改情况来确定哪些文件需要重新编译，从而自动编译所需要的文件并连接目标程序。

【程序 5.2】含有 8 个 C 文件和 3 个头文件的工程 Makefile 实例：

```
all : main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o
        gcc -o all main.o kbd.o command.o display.o \
                insert.o search.o files.o utils.o
main.o : main.c defs.h
        gcc -c main.c
kbd.o : kbd.c defs.h command.h
        gcc -c kbd.c
command.o : command.c defs.h command.h
        gcc -c command.c
display.o : display.c defs.h buffer.h
        gcc -c display.c
insert.o : insert.c defs.h buffer.h
        gcc -c insert.c
search.o : search.c defs.h buffer.h
        gcc -c search.c
files.o : files.c defs.h buffer.h command.h
        gcc -c files.c
utils.o : utils.c defs.h
        gcc -c utils.c
clean :
        rm all main.o kbd.o command.o display.o \
            insert.o search.o files.o utils.o
```

说 明

在程序 5.2 中，反斜杠“\”为换行符，当某一条规则或命令不能在一行中写完时，可以使用它表示换行。可以把程序 5.2 的内容保存为当前目录下的“Makefile”文件或“makefile”文件，或“makefile”文件夹下的文件，然后在该目录下直接输入命令“make”，就可以自动生成可执行文件 all。如果要删除可执行文件和所有的中间目标文件，只要简单地执行一下“make clean”命令就可以了。

另外，在 Makefile 中，目标名称的指定常常有以下惯例(当然也可以不使用这些惯例)：

- all：表示编译所有的内容，是执行 make 时默认的最终目标。
- clean：表示清除所有目标文件。
- distclean：表示清除所有的内容。
- install：表示进行安装的内容。

在程序 5.2 所示的 Makefile 文件中，目标(target)包含如下内容：最终的可执行文件 all 和 8 个中间目标(*.o)；依赖文件(prerequisites)即每个冒号后面的那些.c 文件和.h 文件。每一个.o 文件都有一组依赖文件，而这些.o 文件又是最终目标 all 的依赖文件。依赖关系的实质是说明目标文件由哪些文件生成，换而言之，目标文件是哪些文件更新的结果。在定义好依赖关系后，后续的代码定义了如何生成目标文件的操作系统命令，即生成目标的方法，注意一定要以一个 Tab 键作为它的开头。

说 明

make 并不管命令是怎么工作的，它只管执行所定义的命令。make 会比较 targets 文件和 prerequisites 文件的修改日期，如果 prerequisites 文件的日期比 targets 文件的日期要新，或者 target 不存在，make 就会执行后续定义的命令。另外，程序 5.2 中的 clean 不是一个文件，它只不过是一个动作名字，有点像 C 语言中的 lable 一样，冒号后什么也没有，这样 make 就不会自动去找文件的依赖性，也就不会自动执行其后所定义的命令。要执行其后的命令，就要在 make 命令后显式地指出 clean 这个名字。这样的方法非常有用，可以在一个 Makefile 文件中定义不用的编译或是和编译无关的命令，比如程序的打包或备份等。

在默认方式下，只输入 make 命令，make 便会自动在当前目录下寻找名为“Makefile”或“makefile”的文件，或“makefile”文件夹下的文件。如果找到，它会继续查找 Makefile 文件中的第一个目标。比如在程序 5.2 中，make 会找到 all 这个目标文件，并把这个文件作为最终的目标。如果 all 文件不存在，或是 all 所依赖的 8 个.o 文件中任何一个的修改时间比 all 文件新，make 就会执行后面所定义的命令来生成 all 目标。

如果 all 所依赖的.o 文件也存在，make 会在当前文件中找目标为.o 文件的依赖性，如果找到，则会根据规则生成.o 文件(这有点像一个堆栈的过程)。

当然，C 文件和 H 文件如果存在，make 会生成.o 文件，然后再用.o 文件生成 make 的最终结果，也就是执行文件 all。

这就是整个 make 的依赖性，make 会一层又一层地去找文件的依赖关系，直到最终编译出第一个目标文件。在找寻的过程中，如果出现错误，比如最后被依赖的文件找不到，make 就会直接退出，并报错。而对于所定义的命令的错误，或是编译不成功，make 就不会处理。如果在 make 找到了依赖关系之后，冒号后面的文件不存在，make 仍不工作。

通过上述分析，可以看出像 clean 这样没有被第一个目标文件直接或间接关联时，它后面所定义的命令将不会被自动执行，不过，可以显式使 make 执行。即使用命令 make clean，以此来清除所有的目标文件，并重新编译。

在编程中，如果这个工程已被编译过了，当修改了其中一个源文件时，比如 file.c，根据依赖性，目标 file.o 会被重新编译(也就是在这个依赖性关系后面所定义的命令)，则 file.o 文件也是最新的，即 file.o 文件的修改时间要比 all 要新，所以 all 也会被重新连接了。而如果改变了 command.h、kdb.o、command.o 和 files.o 都会被重新编译，并且 all 会被重新连接。

5.2 Makefile 的书写规则

Makefile 的书写规则包含两个部分，一个是依赖关系，一个是生成目标的方法。

在 Makefile 中，规则的顺序是很重要的，因为 Makefile 中只应该有一个最终目标，其他的目标都是被这个目标所连带出来的，所以一定要让 make 知道最终目标是什么。一般来说，定义在 Makefile 中的目标可能会有很多，但是第一条规则中的目标将被确立为最终的目标。如果

第一条规则中的目标有很多个，那么，第一个目标会成为最终的目标，make 所完成的也就是这个目标。

Makefile 里主要包含了 5 方面的内容：显式规则、隐式规则、变量定义、文件指示和注释。

- 显式规则。显式规则说明了如何生成一个或多个目标。这需要由 Makefile 的书写者显式指出要生成的文件、文件的依赖文件及生成的命令。
- 隐式规则。由于 make 有自动推导的功能，会选择一套默认的方法进行 make，所以隐式的规则可以让开发者比较简略地书写 Makefile，这是由 make 所支持的。
- 变量定义。在 Makefile 中需要定义一系列的变量，一般都是字符串，它类似 C 语言中的宏，当 Makefile 被执行时，其中的变量都会被扩展到相应的引用位置上。
- 文件指示。包括 3 个部分，第一部分是在一个 Makefile 中引用另一个 Makefile，就像 C 语言中的 include 一样包含进来；第二部分是指根据某些情况指定 Makefile 中的有效部分，就像 C 语言中的预编译宏#ifdef 一样；第三部分就是定义一个多行的命令。
- 注释。Makefile 中只有行注释，和 UNIX 的 Shell 脚本一样，其注释符使用井号“#”字符，这就像 C/C++中的双斜杠“//”一样。如果需要在 Makefile 中使用井号“#”字符，可以用反斜杠进行转义，如“\\#”。

注 意

Makefile 文件中的命令必须要以“Tab”键开始。

5.2.1 Makefile 的基本语法规则

任何一种编程语言都有自己的语法格式，Makefile 的语法格式如下：

```
targets : prerequisites
command
...
```

或者是：

```
targets : prerequisites ; command
command
...
```

- targets 是目标文件名，多个文件以空格分开，可以使用通配符。一般来说，Makefile 的目标是一个文件，但也有可能是多个文件。
- prerequisites 是目标所依赖的文件(或依赖目标)。
- command 是命令行，如果它不与“target:prerequisites”在一行，那么，必须以 Tab 键开头，如果和 prerequisites 在同一行，那么可以用分号作为分隔。如果命令太长，可以使用反斜线“\”作为换行符。make 对一行中有多少个字符没有限制。

规则告诉 make 两件事，文件的依赖关系和如何生成目标文件。一般来说，make 会以 UNIX 的标准 Shell，也就是/bin/sh 来执行命令。

下面我们来看一下 5.1.1 小节的程序 5.1 中书写的那个简单的 Makefile：

```
all : main.c foo1.c foo2.c foo3.c
```

```
gcc main.c foo1.c foo2.c foo3.c -o all
```

在这个 Makefile 中，all 就是 Makefile 的最终目标，main.c、foo1.c、foo2.c 和 foo3.c 是目标所依赖的源文件，而只有一个命令“gcc main.c foo1.c foo2.c foo3.c -o all”(以 Tab 键开头)是生成目标的方法。这个规则告诉我们两件事：

(1) 文件的依赖关系。可执行文件 all 依赖于 main.c、foo1.c、foo2.c 和 foo3.c 4 个源文件，如果 main.c、foo1.c、foo2.c 和 foo3.c 中的任何一个文件的修改日期比 all 文件日期新，或者是 all 不存在，那么依赖关系发生。

(2) 如何生成(或更新)all 文件。也就是那个 gcc 命令，其说明了如何生成 all 这个文件，即生成目标的方法。

5.2.2 在规则中使用通配符

如果想定义一系列比较类似的文件，我们很自然地就想起使用通配符。make 支持 3 种通配符：“*”，“?”和“[...]”，这和 UNIX 的 BShell 是相同的。

波浪号(“~”)字符在文件名中也有比较特殊的用途。比如“~/test”，这就表示当前用户的$HOME 目录下的 test 目录。而“~zhangfan/test”则表示用户 zhangfan 的宿主目录下的 test 目录。而在 Windows 或是 MS-DOS 下，用户没有宿主目录，那么波浪号所指的目录则根据环境变量“HOME”而定。

通配符代替了一系列的文件，如“*.c”表示了所有以后缀名为.c 的文件。如果文件名中含有通配符，如“*”，那么可以用转义字符斜杠“\”，如“*”来表示真实的“*”字符，而不是任意长度的字符串。

下面是一个在命令中使用通配符的例子：

```
clean:
rm -f *.o
```

例子的含义是删除所有以.o 为后缀名的文件，这是由操作系统的 Shell 所支持的通配符。

通配符也可以使用在 Makefile 的规则中，比如：

```
print: *.c
lpr -p $?
touch print
```

目标 print 依赖于所有的.c 文件。其中“$?”是一个自动化变量，表示所有比目标新的依赖文件的集合。关于自动化变量，读者可参考本章 5.6.3 小节的内容。

通配符还可以使用在变量中，比如：

```
objects = *.o
```

需要注意的是，这和上面的两种情况不同，这里的*.o 不会展开。objects 变量的值就是“*.o”。Makefile 中的变量其实就是 C/C++中的宏。如果要让通配符在变量中展开，也就是让 objects 的值是所有.o 的文件名的集合，那么，可以这样：

```
objects := $(wildcard *.o)
```

这种用法是通过 Makefile 的关键字“wildcard”来指示的。符号“*”是 Linux 下最常用的通配符之一，读者务必认真体会上面给出的 3 个例子，区分它们之间的异同。

关于另外两个通配符“?”和“[...]”的使用与“*”的使用大同小异，读者在本章后续的内容中可仔细体会。

5.2.3 伪目标

在 5.1.2 小节中程序 5.2 的最后，我们见过一个“clean”的目标，这是一个“伪目标”。伪目标并不是一个文件，而只是一个标签，Makefile 并不生成“clean”这个文件。下面是程序 5.2 中的最后部分代码：

```
clean :
        rm all main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

程序 5.2 的代码中生成了许多的编译文件，比如 main.o、kbd.o、command.o 等，“clean”的作用就是提供一个清除这些编译文件的目标，以备完整地重编译它们而用。

由于伪目标不是文件，所以 make 无法生成它的依赖关系和决定它是否要执行，只有通过显式地指明这个“目标”才能让其生效。需要注意的是，伪目标的取名不能和文件名重名，不然就失去其伪目标的意义了。

当然，为了避免和文件重名的这种情况，我们可以使用一个特殊的标记“.PHONY”来显式地指明一个目标是伪目标，向 make 说明，不管是否有这个文件，这个目标都是伪目标。

```
.PHONY : clean
```

只要有这个声明，不管是否有“clean”文件，要运行“clean”这个目标，只要在命令提示符下输入命令“make clean”即可。于是整个过程可以这样写：

```
.PHONY: clean
clean :
        rm all main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

伪目标一般没有依赖文件，但是，我们也可以为伪目标指定所依赖的文件。伪目标同样可以作为默认目标，只要将它放在第一个。比如，如果 Makefile 需要一口气生成若干个可执行文件，但只想简单地输入一个“make”命令就完事，并且，所有的目标文件都写在一个 Makefile 中，那么便可以使用伪目标这个特性，比如程序 5.3 所示的代码。

【程序 5.3】使用伪目标：

```
all : prog1 prog2 prog3
.PHONY : all
prog1 : prog1.o utils.o
cc -o prog1 prog1.o utils.o
prog2 : prog2.o
cc -o prog2 prog2.o
prog3 : prog3.o sort.o utils.o
cc -o prog3 prog3.o sort.o utils.o
```

我们知道，Makefile 中的第一个目标会被作为其默认目标。在程序 5.3 中声明了一个伪目标 all，其依赖于其他 3 个目标 prog1、prog2 和 prog3。由于伪目标的特性是总是被执行的，所以其依赖的那 3 个目标就总是不如“all”这个目标新，在进行 make 重编译时其他 3 个目标的规则就会总是被决议，也就达到了我们一下生成多个目标的目的。程序 5.3 中，“.PHONY : all”声明了“all”这个目标为“伪目标”。

另外，从程序 5.3 中还可以看出，目标也可以成为依赖。所以，伪目标同样也可成为依赖，如程序 5.4。

【程序 5.4】将伪目标作为依赖文件：

```
.PHONY: cleanall cleanobj cleandiff
cleanall : cleanobj cleandiff
rm program
cleanobj :
rm *.o
cleandiff :
rm *.diff
```

当执行“make clean”命令时，将清除所有要被清除的文件。“cleanobj”和“cleandiff”这两个伪目标有点像子程序。当然也可以输入“make cleanall”、“make cleanobj”和“make cleandiff”命令来达到清除不同种类文件的目的。

5.2.4 多目标

前面已向读者提到过 Makefile 规则中的目标可以不止一个，它支持多目标，当 Makefile 中的多个目标同时依赖于同一个文件，并且其生成的命令大体类似，就能把它们合并起来。

当然，多个目标的生成规则的执行命令是同一个可能会给我们带来麻烦，不过我们可以使用一个自动化变量“$@”(关于自动化变量，将在后面讲述)，这个变量意味着目前规则中所有的目标的集合，下面来看一个例子，如程序 5.5。

【程序 5.5】同时定义多个目标：

```
bigoutput littleoutput : text.g
generate text.g -$(subst output,,$@) >; $@
```

程序 5.5 的规则等价于：

```
bigoutput : text.g
generate text.g -big >; bigoutput
littleoutput : text.g
generate text.g -little >; littleoutput
```

其中，-$(subst output,,$@)中的“$”表示执行一个 Makefile 的函数，函数名为 subst，后面的为参数(关于函数，也将在后面讲述)。subst 函数是截取字符串的意思，“$@”表示目标的集合，就像一个数组，“$@”依次取出目标并执行命令。

5.2.5 自动生成依赖性

在 Makefile 中，生成目标的依赖关系中可能会需要包含一系列的头文件，比如，如果在

main.c 的源代码中有一句“#include "defs.h"”，那么依赖关系应该是：

```
main.o : main.c defs.h
```

但是，如果是一个比较大型的工程，就必须清楚哪些 C 文件包含了哪些头文件，并且在加入或删除某些头文件时，也需要小心翼翼地修改 Makefile，这是一件相当繁琐的工作。为了避免这种繁琐而又容易出错的事情，我们可以使用 GNU 的 C/C++编译器的“-MM”参数选项，使其自动找寻源文件中包含的头文件，并生成一个依赖关系。例如，如果执行下面的命令：

```
gcc -MM main.c
```

其输出是：

```
main.o : main.c defs.h
```

于是由编译器自动生成依赖关系，这样一来，就不必再手动书写若干文件的依赖关系了。

实际上，大多数的 C/C++编译器都支持这种自动生成文件依赖关系的功能。它们和 GNU 的 C/C++编译器的区别是参数选项为“-M”，而不是“-MM”。例如，我们还可以使用下面的命令：

```
cc -M main.c
```

其输出同样是：

```
main.o : main.c defs.h
```

需要提醒读者的是，GNU 的 C/C++编译器必须要用“-MM”参数，不然，“-M”参数会把一些标准库的头文件也包含进来。例如，命令：

```
gcc -M main.c
```

其输出是：

```
main.o: main.c defs.h /usr/include/stdio.h /usr/include/features.h \
/usr/include/sys/cdefs.h /usr/include/gnu/stubs.h \
/usr/lib/gcc-lib/i486-suse-linux/2.95.3/include/stddef.h \
/usr/include/bits/types.h /usr/include/bits/pthreadtypes.h \
/usr/include/bits/sched.h /usr/include/libio.h \
/usr/include/_G_config.h /usr/include/wchar.h \
/usr/include/bits/wchar.h /usr/include/gconv.h \
/usr/lib/gcc-lib/i486-suse-linux/2.95.3/include/stdarg.h \
/usr/include/bits/stdio_lim.h
```

那么，编译器的这个功能如何与 Makefile 联系在一起呢？因为这样一来，Makefile 也要根据这些源文件重新生成，让 Makefile 依赖于源文件吗？这个功能并不现实，不过我们可以用其他手段来迂回地实现这一功能。GNU 组织建议把编译器为每一个源文件的自动生成的依赖关系放到一个文件中，例如为每一个“name.c”的文件都生成一个“name.d”的 Makefile 文件，.d 文件中就存放着对应.c 文件的依赖关系。

于是，可以写出.c 文件和.d 文件的依赖关系，并让 make 自动更新或生成.d 文件，并把其

包含在最终的 Makefile 中，这样，就可以自动化地生成每个文件的依赖关系了。

5.3 Makefile 的命令

Makefile 中的命令和操作系统 Shell 的命令行是一致的。make 会按顺序一条一条地执行命令，每条命令必须以“Tab”键开始，除非命令是紧跟在依赖规则后面的分号后的。在命令行之间的空格或者空行会被忽略，但是如果该空格或空行是以“Tab”键开始的，make 便会认为其是一个空命令。

在 Linux 下有各种不同的 Shell，但是 make 的命令默认是被“/bin/sh”——UNIX 的标准 Shell 解释执行的，除非特别指定一个其他的 Shell。

当依赖目标新于生成目标时，也就是当规则的最终目标需要被更新时，make 会一条一条地执行其后的命令。如果要让上一条命令的结果应用在下一条命令，需要使用分号分隔这两条命令。比如第一条命令是 cd 命令，希望第二条命令在 cd 之后的基础上运行，那么就不能把这两条命令写在两行上，而应该把这两条命令写在一行上，并用分号分隔，比如下面这段代码：

```
exec:
cd /home/zhangfan                #进入/home/zhangfan 目录
pwd                              #打印当前目录
```

执行“make exec”命令后，会进入/home/zhangfan 目录，但是 pwd 命令打印出的结果仍是当前的 Makefile 目录，如果改写成下面这样：

```
exec:
cd /home/zhangfan; pwd           #进入/home/zhangfan 目录后，打印当前目录
```

执行“make exec”命令后，系统进入/home/zhangfan 目录，并且 pwd 打印出的结果是“/home/zhangfan”。

每当命令执行完毕后，make 会自动检测它们的返回码，如果命令返回成功，那么 make 会继续执行下一条命令，直到规则中的所有命令都成功返回。如果规则中的某个命令出错了(命令退出代码非零)，那么 make 就会终止执行当前规则，这将有可能终止所有规则的执行。

但有些时候，命令的出错并不表示规则就是错误的。比如在上面示例的代码中，如果/home/zhangfan 目录是存在的，那么 cd 命令就成功执行；如果目录不存在，那么就出错了。但是在某些情况下是不希望因为 cd 命令的出错而终止规则的运行的。

为了做到这一点，忽略命令的出错，我们可以在 Makefile 的命令行前加一个减号“-”(在“Tab”键之后)，标记为不管命令是否出错都认为是成功的。于是，可以将上面的例子改写成：

```
exec:
- cd /home/zhangfan              #进入/home/zhangfan 目录，并忽略错误
pwd                              #打印当前目录
```

还有一个全局的办法是，为 make 加上“-i”或是“--ignore-errors”参数，那么，Makefile 中所有命令都会忽略错误。而如果一个规则是以“.IGNORE”作为目标的，那么，这个规则中

的所有命令将会忽略错误。这些是不同级别的防止命令出错的方法，程序员可以根据不同的需要进行设置。

另一个参数是“-k”或“--keep-going”，这个参数的意思是，如果某个规则中的命令出错了，那么就终止该规则的执行，但继续执行其他规则。

另外，make 通常会把要执行的命令行在执行前输出到屏幕上。当用“@”字符在命令行前，这个命令将不被 make 显示出来，最具代表性的例子是可以用这个功能来向屏幕显示一些信息。如：

```
@echo 正在编译 XXX 模块......
```

当 make 执行时，会输出“正在编译 XXX 模块......”字串，但不会输出命令，如果没有“@”，那么，make 将输出：

```
echo 正在编译 XXX 模块......
正在编译 XXX 模块......
```

如果 make 执行时，带入 make 参数“-n”或“--just-print”，那么它只是显示命令，但不会执行命令，这个功能有利于调试 Makefile，看看书写的命令执行起来是什么样子的或是什么顺序的。而 make 参数“-s”或“--slient”则是全面禁止命令的显示。

5.4 变量

Makefile 的变量就像是 C/C++语言中的宏，代表了一个文本字串，在 Makefile 执行的时候会自动展开在所使用的地方。与 C/C++的宏所不同的是，Makefile 变量的值是可以改变的。在 Makefile 中，变量可以使用在目标、依赖目标、命令或是 Makefile 的其他部分中。

5.4.1 变量的基础

变量的命名可以包含字符、数字、下划线(可以是数字开头)，但不能含有“:”、“#”、“=”或是空字符(空格、回车等)。变量是大小写敏感的，“foo”、“Foo”和“FOO”是 3 个不同的变量名。传统的 Makefile 的变量名是全大写的命名方式，但笔者推荐使用大小写搭配的变量名，如：MakeFlags。这样可以避免和系统的变量冲突，而发生意外的事情。

变量在声明时需要给予初值，而在使用时，需要在变量名前加上“$”符号，但最好用小括号“()”或是花括号“{ }”把变量给括起来。如果要使用真实的“$”字符，那么需要用“$$”来表示。程序 5.6 是一个关于 Makefile 变量的基础性的例子。

【程序 5.6】Makefile 中使用变量：

```
objects = program.o foo.o utils.o
program : $(objects)
cc -o program $(objects)
$(objects) : defs.h
```

变量会在使用它的地方精确地展开，就像 C/C++中的宏一样，程序 5.6 中的变量展开后

得到：

```
objects = program.o foo.o utils.o
program : program.o foo.o utils.o
cc -o program program.o foo.o utils.o
program.o foo.o utils.o : defs.h
```

由此可见，Makefile 的变量就是一个“替代”的原理。

提 示

在使用变量时加上括号完全是为了更加安全地使用它，在上面的例子中，也可以不给变量加上括号，但建议读者在学习 Makefile 时养成良好的习惯，使用变量时一直加括号。

5.4.2 赋值变量

除了像程序 5.6 所示的代码那样，使用“＝”符号给变量赋值外，Makefile 还支持变量的嵌套定义，例如：

```
foo=$(bar)
bar=$(ugh)
ugh=Huh?
all:
echo $(foo)
```

执行“make all”将会输出变量 foo 的值是“Huh?”($(foo)的值是$(bar)，$(bar)的值是$(ugh)，$(ugh)的值是“Huh?”)。

Makefile 变量的这个特性有利有弊，好处是可以把变量的真实值推到后面来定义，如：

```
CFLAGS=$(include_dirs) -O
include_dirs=-Ifoo -Ibar
```

当“CFLAGS”在命令中被展开时，会是“-Ifoo -Ibar -O”。但这种形式也有弊端，那就是递归定义，如：

```
CFLAGS = $(CFLAGS) -O
```

或者：

```
A=$(B)
B=$(A)
```

这会让 make 陷入无限的变量展开过程中去，当然，make 有能力检测出这样的定义，并会报错。但是为了避免上面的这种问题，可以使用 make 中的另一种用变量来定义变量的方法。这种方法使用的是“:=”操作符，如：

```
x :=foo
y :=$(x) bar
x :=later
```

其等价于：

```
y :=foo bar
x :=later
```

使用的是“:=”操作符，使得前面的变量不能使用后面的变量，只能使用前面已定义好了的变量。如果是这样：

```
y :=$(x) bar
x :=foo
```

那么，y 的值是“bar”，而不是“foo bar”。

还有一个很有用的操作符是“?=”，先看示例：

```
FOO ?=bar
```

其含义是，如果变量 FOO 没有被定义过，那么变量 FOO 的值就是“bar”，如果 FOO 先前被定义过，那么这条语句将什么也不做，其等价于：

```
ifeq ($(origin FOO), undefined)
FOO=bar
endif
```

这段代码中使用了 ifeq 条件判断语句和 origin 函数调用。origin 函数返回变量 FOO 的出处，ifeq 则判断两个参数是否相等。关于这些，将在后面的内容向读者介绍。

另外一种赋值操作符是“+=”，它可以给变量追加值，如：

```
objects=main.o foo.o bar.o utils.o
objects +=another.o
```

于是，$(objects)值变成：“main.o foo.o bar.o utils.o another.o”(another.o 被追加进去了)，使用“+=”操作符，可以等效为下面的形式：

```
objects=main.o foo.o bar.o utils.o
objects:=$(objects) another.o
```

所不同的是，使用“+=”更为简洁。

如果变量之前一没有定义过，那么，“+=”会自动变成“=”，如果前面有变量定义，那么“+=”会继承于前次操作的赋值符。如果前一次的是“:=”，那么“+=”会以“:=”作为其赋值符，如：

```
variable:=value
variable+=more
```

等价于：

```
variable:=value
variable:=$(variable) more
```

5.4.3 define 关键字

还有一种设置变量值的方法是使用 define 关键字。使用 define 关键字设置变量的值可以有

换行，这有利于定义一系列的命令。

define 指示符后面跟的是变量的名字，而另起一行定义变量的值，定义是以 endef 关键字结束。其工作方式和“=”操作符一样。变量的值可以包含函数、命令、文字，或是其他变量。

因为命令需要以“Tab”键开头，所以如果用 define 定义的命令变量中没有以“Tab”键开头，那么 make 就不会把其认为是命令。下面的这个示例展示了 define 的用法：

```
define two-lines
echo foo
echo $(bar)
endef
```

这段代码将变量 two-lines 定义为两条命令 echo foo 和 echo $(bar)，这种定义形式能使代码更加紧凑和优美。

5.4.4 override 指示符

有的变量是通过 make 的命令行参数进行设置的，那么 Makefile 将忽略这个变量的赋值。如果想在 Makefile 中设置这类参数的值，那么可以使用“override”指示符。其语法是：

```
override <variable>=<value>
override <variable>:=<value>
```

当然也可以追加值操作符：

```
override <variable>+=<more text>
```

在 define 关键字中也同样可以使用 override 指示符，如：

```
override define foo
bar
endef
```

5.4.5 目标变量和模式变量

上面介绍的在 Makefile 中定义的变量都是全局变量，在整个文件中都可以访问这些变量。但是，自动化变量除外，如“$<”等这种自动化变量就属于“规则型变量”，这种变量的值依赖于规则的目标和依赖目标的定义。

当然，同样可以为某个目标设置局部变量，这种变量被称为目标变量(Target-specific Variable)，它可以和全局变量同名，因为它的作用范围只在这条规则及连带规则中，所以其值也只在作用范围内有效。而不会影响规则链以外的全局变量的值。

说明

实际上，在 Makefile 中，并没有“全局变量”和“局部变量”的概念，但目标变量的这种特性的确与 C/C++中的全局变量和局部变量十分相像。读者完全可以将它们理解为 Makefile 的“局部变量”。

其语法是：

```
<target ...> : <variable-assignment>
<target ...> : overide <variable-assignment>
```

<target ...>为目标序列，<variable-assignment>可以是前面讲过的各种赋值表达式，如"="、":="、"+="或是"？="。第二个语法是针对 make 命令行参数带入的变量，或是系统环境变量。

这个特性非常有用，当我们设置了这样一个变量，这个变量会作用到由这个目标所引发的所有的规则中去。如程序 5.7 所示的代码。

【程序 5.7】Makefile 中使用目标变量 L：

```
prog : CFLAGS = -g                    #定义目标变量 CFLAGS，其值为-g
prog : prog.o foo.o bar.o
$(CC) $(CFLAGS) prog.o foo.o bar.o          #引用变量 CC 和 CFLAGS
prog.o : prog.c
$(CC) $(CFLAGS) prog.c
foo.o : foo.c
$(CC) $(CFLAGS) foo.c
bar.o : bar.c
$(CC) $(CFLAGS) bar.c
```

在这个示例中，不管全局的$(CFLAGS)的值是什么，在 prog 目标，以及其所引发的所有规则中(prog.o,foo.o 和 bar.o 的规则)，$(CFLAGS)的值都是"-g"。将程序 5.7 的代码展开，读者就可以一目了然了，如下：

```
prog : prog.o foo.o bar.o
gcc -g prog.o foo.o bar.o
prog.o : prog.c
gcc -g prog.c
foo.o : foo.c
gcc -g foo.c
bar.o : bar.c
gcc -g bar.c
```

在 GNU 的 make 中，还支持模式变量(Pattern-specific Variable)。模式变量是目标变量的一种，它的特点是可以给定一种"模式"，把变量定义在符合这种模式的所有目标中。

同样，模式变量的语法和目标变量一样：

```
<pattern ...> : <variable-assignment>
<pattern ...> : override <variable-assignment>
```

<pattern...>是模式序列，可以是多种模式，override 同样是针对系统环境传入的变量，或是 make 命令行指定的变量。

我们知道，make 的模 x 式一般是至少含有一个"%"的，所以，可以以如下方式给所有以.o 结尾的目标定义目标模式变量：

```
%.o : CFLAGS = -O
```

5.5 常用函数调用

Makefile 可以调用函数，从而让 Makefile 的书写变得更为灵活和简洁。函数调用后，返回值可以当作变量来使用，并且函数的调用也很像变量的使用，也是以“$”来标识的，其语法如下：

```
$(<function> <arguments>)
```

或是：

```
${<function> <arguments>}
```

<function>是函数名，<arguments>是函数的参数，参数间以逗号隔开，而函数名和参数之间以空格分隔。函数调用以“$”开头，以圆括号或花括号把函数名和参数括起来。函数中的参数可以使用变量，为了风格的统一，函数和变量的括号最好一样，如使用“$(subst a,b,$(x))”这样的形式，而不是“$(subst a,b,${x})”的形式。

5.5.1 字符串处理函数

1. 字符串替换函数 subst

格式：

```
$(subst <from>,<to>,<text>)
```

功能：把字串<text>中的<from>字符串替换成<to>。
返回：函数返回被替换过后的字符串。
示例：

```
$(subst ee,EE,feet on the street)
```

作用是把字符串“feet on the street”中的“ee”替换成“EE”，返回结果是：

```
fEEt on the strEEt
```

2. 模式字符串替换函数 patsubst

格式：

```
$(patsubst <pattern>,<replacement>,<text>)
```

功能：查找<text>中的单词(单词以“空格”、“Tab”、“回车”或“换行”分隔)是否符合模式<pattern>，如果匹配的话，则以<replacement>模式替换。这里，<pattern>可以包括通配符“%”，用来表示任意长度的字串。如果<replacement>中也包含“%”，那么，<replacement>中的这个“%”将是<pattern>中的那个“%”所代表的字串(可以用“\”来转义，以“\%”来表示真实含义的“%”字符)。

返回：函数返回被替换过后的字符串。

示例：

```
$(patsubst %.c,%.o,x.c.c bar.c)
```

把字串“x.c.c bar.c”符合模式“%.c”的单词替换成“%.o”，函数返回结果是：

```
x.c.o bar.o
```

3. 查找字符串函数 findstring

格式：

```
$(findstring <find>,<in>)
```

功能：在字符串<in>中查找<find>字符串。

返回：如果找到，则返回<find>字符串，否则返回空字符串。

示例：

```
$(findstring a,a b c)
$(findstring a,b c)
```

第一个函数返回“a”字符串，第二个返回“ ”字符串(空字符串)。

4. 过滤函数 filter

格式：

```
$(filter <pattern...>,<text>)
```

功能：以<pattern>模式过滤<text>字符串中的单词，保留符合模式<pattern>的单词。可以有多个模式。

返回：返回符合模式<pattern>的字符串。

示例：

```
sources:=foo.c bar.c baz.s ugh.h
foo: $(sources)
cc $(filter %.c %.s,$(sources)) -o foo
```

$(filter %.c %.s,$(sources))返回的值是“foo.c bar.c baz.s”。

5. 反过滤函数 filter-out

格式：

```
$(filter-out <pattern...>,<text>)
```

功能：以<pattern>模式过滤<text>字符串中的单词，去除符合模式<pattern>的单词。可以有多个模式。

返回：返回不符合模式<pattern>的字串。

示例：

```
objects=main1.o foo.o main2.o bar.o
mains=main1.o main2.o
```

```
$(filter-out $(mains),$(objects))
```

返回值是“foo.o bar.o”。

6. 排序函数 sort

格式：

```
$(sort <list>)
```

功能：将字符串<list>中的单词按照字母排序(升序)进行排序，先比较单词的首字母，首字母相同，则比较下一个字母，以此类推。当遇到相同的单词时，sort 函数会去自动删除它们。

返回：返回排序后的字符串。

示例一：

```
$(sort lose foo bar lost)
```

返回值为：

```
bar foo lose lost
```

示例二：

```
$(sort programing linux c programing)
```

返回值为：

```
c linux programing
```

7. 取单词函数 word

格式：

```
$(word <n>,<text>)
```

功能：从字符串<text>中取出第<n>个单词，单词计数从 1 开始。

返回：返回字符串<text>中的第<n>个单词。如果<n>值比<text>中的单词数要大，则返回空字符串。

示例一：

```
$(word 2, programing linux c programing)
```

返回值是：

```
linux
```

示例二：

```
$(word 5, programing linux c programing)
```

返回值为“ ”(空字符串)。

8. 取单词串函数 wordlist

格式：

```
$(wordlist <s>,<e>,<text>)
```

功能：从字符串<text>中取出从<s>开始到<e>的单词串，<s>和<e>是一个数字。

返回：返回字符串<text>中从<s>到<e>的单词字串。如果<s>比<text>中的单词数要大，那么返回空字符串。如果<e>大于<text>的单词数，那么返回从<s>开始，到<text>结束的单词串。单词计数从 1 开始。

示例一：

```
$(wordlist 2, 4, I like linux c programing)
```

返回值是：

```
like linux c
```

示例二：

```
$(wordlist 6, 8, I like linux c programing)
```

返回值为“ ”(空字符串)。

示例三：

```
$(wordlist 2, 8, I like linux c programing)
```

返回值是:

```
like linux c programing
```

9. 单词个数统计函数 words

格式：

```
$(words <text>)
```

功能：统计<text>字符串中的单词个数，计数从 1 开始。

返回：返回<text>中的单词数。

示例：

```
$(words, I like linux c programing)
```

返回值是“5”。

提 示

Makefile 中的很多函数调用是可以嵌套的，这很像 C/C++中的函数调用。通过嵌套调用函数，使代码更加简洁高效，这通常是程序员高手们所喜好的风格。

比如在上面讲到的 words 函数，可以这样来嵌套调，以实现不同的功能，比如我们要取<text>字符串中的最后一个单词，可以这样来写：

```
$(word $(words<text>),<text>)
```

10. 首单词函数 firstword

格式：

```
$(firstword <text>)
```

功能：取字符串<text>中的第一个单词。
返回：返回字符串<text>的第一个单词。
示例：

```
$(firstword I like linux c programing)
```

返回值是单词“I”。
这个函数也可以用 word 函数来实现：

```
$(word 1,<text>)
```

以上是所有的字符串操作函数，如果搭配混合使用，可以完成比较复杂的功能。下面举一个现实中应用的例子。make 使用“VPATH”变量来指定“依赖文件”的搜索路径。于是，可以利用这个搜索路径来指定编译器对头文件的搜索路径参数 CFLAGS，如：

```
override CFLAGS += $(patsubst %,-I%,$(subst :, ,$(VPATH)))
```

如果“$(VPATH)”值是 src:../headers，那么“$(patsubst %,-I%,$(subst :, ,$(VPATH)))”将返回“-Isrc -I../headers”，这正是 cc 或 gcc 搜索头文件路径的参数。

5.5.2 文件名操作函数

本小节将向读者介绍的函数主要是用来处理文件名的，每个函数的参数字符串都会被当作一个或是一系列的文件名来对待。

1. 取目录函数 dir

格式：

```
$(dir <names...>)
```

功能：从文件名序列(一个或多个文件名)<names>中取出目录部分。目录部分是指最后一个斜杠“/”之前的部分，如果没有斜杠，则返回“./”。
返回：返回文件名序列<names>的目录部分。
示例：

```
$(dir usr/src/linux-2.4/Makefile hello.c)
```

返回值是“usr/src/linux-2.4 ./”。

2. 取文件名函数 notdir

格式：

```
$(notdir <names...>)
```

功能：从文件名序列(一个或多个文件名)<names>中取出非目录部分。非目录部分是指最后一个斜杠“/”之后的部分，即文件名。

返回：返回文件名序列<names>的非目录部分。
示例：

```
$(notdir usr/src/linux-2.4/Makefile hello.c)
```

返回值是“Makefile hello.c”。

3. 取后缀名函数 suffix

格式：

```
$(suffix <names...>)
```

功能：从文件名序列<names>中取出各个文件名的后缀名。
返回：返回文件名序列<names>的后缀名序列，如果文件没有后缀名，则返回空字符串。
示例：

```
$(suffix usr/src/linux-2.4/Makefile hello.c foo.s)
```

返回值是“ .c .s”(第一个为空字符)。

4. 名称取前缀函数 basename

格式：

```
$(basename <names...>)
```

功能：从文件名序列<names>中取出各个文件名的前缀部分。
返回：返回文件名序列<names>的前缀序列，如果文件没有前缀，则返回空字符串。
示例：

```
$(basename usr/src/linux-2.4/kernel/exit.c hello.o home/hacks)
```

返回值是“usr/src/linux-2.4/kernel/exit hello home/hacks”。

5. 加后缀函数 addsuffix

格式：

```
$(addsuffix <suffix>,<names...>)
```

功能：把后缀<suffix>加到<names>中的每个单词后面。
返回：返回已加后缀的文件名序列。
示例：

```
$(addsuffix .c,foo bar hello)
```

返回值是“foo.c bar.c hello.c”。

6. 加前缀函数 addprefix

格式：

```
$(addprefix <prefix>,<names...>)
```

功能：把前缀<prefix>加到<names>中的每个单词前面。
返回：返回加过前缀的文件名序列。
示例：

```
$(addprefix usr/src/linux-2.4/kernel/,exit.c time.c)
```

返回值是“usr/src/linux-2.4/kernel/exit.c usr/src/linux-2.4/kernel/ time.c”。

7. 连接函数 join

格式：

```
$(join <list1>,<list2>)
```

功能：把<list2>中的每个单词对应地插入到<list1>各个单词的后面。大多数情况下，两个字符串中的单词个数并不相等，那么多出来的单词将会被复制到新字符串中。
返回：返回连接过后的字符串。
示例一：

```
$(join aaa bbb , 111 222 333)
```

返回值是“aaa111 bbb222 333”。
示例二：

```
$(join aaa bbb ccc, 111 222)
```

返回值是“aaa111 bbb222 ccc”。

5.5.3 循环函数

foreach 函数是用来做循环用的，Makefile 中的 foreach 函数几乎是仿照 UNIX 标准 Shell(/bin/sh)中的 for 语句，或是 C-Shell(/bin/csh)中的 foreach 语句而构建的。它的语法是：

```
$(foreach <var>,<list>,<text>)
```

这个函数的意思是把参数<list>中的单词逐一取出放到参数<var>所指定的变量中，然后再执行<text>所包含的表达式。每一次<text>会返回一个字符串，循环过程中，<text>的所返回的每个字符串会以空格分隔，最后当整个循环结束时，<text>所返回的每个字符串所组成的整个字符串(以空格分隔)将会是 foreach 函数的返回值。所以，<var>最好是一个变量名，<list>可以是一个表达式，而<text>中一般会使用<var>这个参数来依次枚举<list>中的单词。例如：

```
names := a b c d
files := $(foreach n,$(names),$(n).o)
```

上面的例子中，$(name)中的单词会被依次取出，并存到变量“n”中，“$(n).o”每次根据

“$(n)”计算出一个值，这些值以空格分隔，最后作为 foreach 函数的返回值，所以，$(files)的值是“a.o b.o c.o d.o”。

> **注 意**
>
> foreach 中的<var>参数是一个临时的局部变量，foreach 函数执行完后，参数<var>的变量将不再起作用，其作用域只在 foreach 函数当中。

5.5.4 条件判断函数

在向读者介绍 if 函数之前，有必要先介绍 GNU 的 make 所支持的条件判断语句。

使用条件判断，可以让 make 根据运行时的不同情况选择不同的执行分支。条件表达式可以是比较变量的值，或是比较变量和常量的值。它的语法也很像 C/C++中的 if 条件判断语句，不同的是，Makefile 的条件判断语句所支持的关键字有 4 个，它们的含义如表 5.2 所示。

表 5.2 条件判断关键字

关 键 字	典型表达式	含 义
ifeq	ifeq (<arg1>, <arg2>)	比较参数 arg1 和 arg2 的值是否相等，相等时表达式为真
ifneq	ifneq(<arg1>, <arg2>)	比较参数 arg1 和 arg2 的值是否相等，不相等时表达式为真
ifdef	ifdef <variable-name>	如果变量<variable-name>的值非空，则表达式为真。<variable-name>可以是一个函数的返回值。ifdef 只是测试一个变量是否有值，其并不会把变量扩展到当前位置
ifndef	ifndef <variable-name>	如果变量<variable-name>的值为空，则表达式为真

> **注 意**
>
> make 是在读取 Makefile 时就计算条件表达式的值，并根据条件表达式的值来选择语句，所以，最好不要把自动化变量(如“$@”等)放入到条件表达式中，因为自动化变量是在运行时才有的。而且，为了避免混乱，make 不允许把整个条件语句分成两部分放在不同的文件中。

下面要介绍的 if 函数很像 make 所支持的条件判断语句，if 函数的语法是：

```
$(if <condition>,<then-part>)
```

或是：

```
$(if <condition>,<then-part>,<else-part>)
```

可见，if 函数可以包含“else”部分，也可以不包含。即 if 函数的参数可以是两个，也可以是三个。<condition>参数是 if 的表达式，如果其返回的为非空字符串，那么这个表达式就相当于返回真，于是，<then-part>会被计算，否则<else-part>会被计算。

而 if 函数的返回值是，如果<condition>为真(非空字符串)，那么<then-part>会是整个函数的返回值，如果<condition>为假(空字符串)，那么<else-part>会是整个函数的返回值，此时如果

<else-part>没有被定义，整个函数返回空字符串。所以，<then-part>和<else-part>只会有一个被执行。

5.5.5 其他常用函数

1. call 函数

call 函数是唯一一个可以用来创建新的参数化的函数。程序员可以写一个非常复杂的表达式，这个表达式中，可以定义许多参数，然后使用 call 函数来向这个表达式传递参数。其语法是：

```
$(call <expression>,<parm1>,<parm2>,<parm3>...)
```

当 make 执行这个函数时，<expression>参数中的变量，如$(1)、$(2)、$(3)等，会被参数<parm1>、<parm2>、<parm3>依次取代。而<expression>的返回值就是 call 函数的返回值。例如：

```
reverse=$(1) $(2)
foo=$(call reverse,a,b)
```

那么，foo 的值就是“a b”。当然，参数的次序是可以自定义的，不一定是顺序的，例如：

```
reverse = $(2) $(1)
foo = $(call reverse,a,b)
```

此时的 foo 的值就是“b a”。

2. origin 函数

origin 函数不像其他的函数，它并不操作变量的值，只是告诉程序变量是从哪里来的。其语法是：

```
$(origin <variable>)
```

注 意

<variable>是变量的名字，不应该是引用。所以最好不要在<variable>中使用“$”字符。

origin 函数会返回变量<variable>的“出生情况”，表 5.3 所列是 origin 函数各种不同的返回值。

表 5.3 origin 函数的返回值

返 回 值	含 义
Undefined	如果<variable>从来没有定义过，origin 函数返回这个值
Default	如果<variable>是一个默认的定义，比如“CC”这个变量
environment	如果<variable>是一个环境变量，并且当 Makefile 被执行时，“-e”参数没有被打开
file	如果<variable>变量被定义在 Makefile 中
command line	如果<variable>变量是被命令行定义的

(续表)

返　回　值	含　　义
override	如果<variable>变量是被 override 指示符重新定义的
automatic	如果<variable>是一个命令运行中的自动化变量

origin 函数的返回信息对于编写 Makefile 是非常有用的，比如，假设有一个 Makefile 中包含了一个定义文件 Make.def，在 Make.def 中定义了一个变量“bletch”，而系统环境中可能也存在一个与之同名的环境变量“bletch”。此时，就需要判断“bletch”是否是环境变量，如果变量来源于环境，就把它重定义了，如果来源于 Make.def 或是命令行等非环境的，那么就不需要重新定义它。于是，在 Makefile 中可以这样写：

```
ifdef bletch                                #是否存在变量 bletch
ifeq "$(origin bletch)" "environment"       #bletch 是否为环境变量
bletch=barf, gag, etc.                      #如果是，将它重新定义
endif
endif
```

读者也许会问，使用 override 关键字不就可以重新定义环境中的变量了吗？为什么需要使用这样的步骤？是的，使用 override 的确可以达到这样的效果，可是 override 过于粗暴，它同时会把从命令行定义的变量也覆盖，而这里只想重新定义环境传来的变量，不想重新定义命令行传来的变量。

3. shell 函数

顾名思义，shell 函数就是执行操作系统 Shell 命令的函数，它的参数就是操作系统的 Shell 命令。也就是说，shell 函数把执行操作系统命令后的输出作为函数返回。于是，可以用操作系统命令及字符串处理命令 awk，sed 等来生成一个变量，这一点在程序员编写 Makefile 时是非常有用的，比如这样来定义变量：

```
contents := $(shell cat foo)        # 变量 content 的值定义为打印文件 foo 的内容
files := $(shell echo *.c)          # 变量 files 的值定义为显示所有以.c 为后缀的文件的名称
```

注 意

这个函数会新生成一个 Shell 程序来执行命令，所以要注意其运行性能，如果 Makefile 中有一些比较复杂的规则，并大量使用了这个函数，那么对于系统的性能是有害的。特别是 Makefile 的隐晦的规则可能会让 shell 函数执行的次数比想像要多得多。因此，应该控制 shell 函数的使用。

4. 控制 make 的函数

make 提供了一些函数来控制 make 的运行。通常，需要检测一些运行 Makefile 时的信息，并根据这些信息来决定是让 make 继续执行，还是停止。常用的是 error 和 warning 函数。error 函数的格式如下：

```
$(error <text ...>)
```

函数产生一个致命的错误，<text ...>是错误信息。error 函数不会在一开始被使用时就产生错误信息，所以如果把它定义在某个变量中，并在后续的脚本中使用这个变量，那么也是可以的。例如：

```
ifdef ERROR_001
$(error error is $(ERROR_001))
endif
```

代码的含义是，如果之前定义了变量 ERROR_001，则产生错误信息，并引用这个变量。

warning 函数的格式如下：

```
$(warning <text ...>)
```

它很像 error 函数，<text ...>表示警告信息，不同的是，warning 函数并不会让 make 退出，只是输出一段警告信息，而 make 继续执行。

5.6 隐式规则

顾名思义，隐式规则即 Makefile 预先约定好了的不用再写出来的规则。比如把.c 文件编译成.o 文件这一规则，不用写出来，make 便会自动推导，并生成我们需要的.o 文件。隐式规则会使用一些系统变量，可以改变这些系统变量的值来定制隐式规则运行时的参数。如系统变量“CFLAGS”可以控制编译时的编译器参数。还可以通过模式规则的方式自定义隐式规则。

了解隐式规则，可以让其更好地服务，也会让我们知道一些约定俗成的规则，而不至于在运行 Makefile 时出现一些莫名其妙的错误。当然，有时候隐式规则也会给程序员造成不小的麻烦。只有真正了解它，才能更好地使用它。

5.6.1 隐式规则举例

如果要使用隐式规则来生成目标，就不需要写出这个目标的规则，make 会试图去自动推导产生这个目标的规则和命令，如果 make 可以自动推导生成这个目标的规则和命令，那么这个行为就是隐式规则的自动推导。当然，隐式规则是 make 事先约定好的一些东西。例如，有下面的 Makefile 文件：

```
foo : foo.o bar.o
cc –o foo foo.o bar.o $(CFLAGS) $(LDFLAGS)
```

可以看到，这个 Makefile 中并没有写如何生成 foo.o 和 bar.o 这两目标的规则和命令。因为 make 的隐式规则功能会自动推导这两个目标的依赖目标和生成命令。

make 会在自己的隐式规则库中寻找可以使用的规则，如果找不到，就会报错。在上面的那个例子中，make 调用的隐式规则是把.o 目标的依赖文件置成.c，并使用 C 的编译命令“cc –c $(CFLAGS)”来生成.o 的目标文件。也就是说，完全没有必要写出下面的两条规则：

```
foo.o : foo.c
cc –c foo.c $(CFLAGS)
bar.o : bar.c
cc –c bar.c $(CFLAGS)
```

因为这已经是“约定”好了的规则，make 使用约定好的 C 编译器“cc”生成.o 文件，这就是隐式规则。

当然，如果程序员为.o 文件书写了自己的规则，那么 make 就不会自动推导并调用隐式规则，它会按照程序员写好的规则忠实地执行。

另外，在 make 的隐式规则库中，每一条隐式规则在库中都有其顺序，越靠前的规则越经常被使用，所以，这会导致有些时候即使显式地指定了目标依赖，make 也不会理会。如下面的这条规则(没有命令)：

```
foo.o : foo.p
```

依赖文件 foo.p(Pascal 程序的源文件)有可能变得没有意义。如果目录下存在了 foo.c 文件，那么隐式规则一样会生效，并会通过 foo.c 调用 C 的编译器生成 foo.o 文件。因为在隐式规则中，Pascal 的规则出现在 C 的规则之后，所以，make 找到可以生成 foo.o 的 C 的规则就不再寻找下一条规则了。如果确实不希望任何隐式规则推导，就不要只写出依赖规则，而不写命令。

有些时候，一个目标可能被一系列的隐式规则所作用。例如，一个.o 的文件生成，可能会是先被 Yacc 的.y 文件先生成.c，然后再被 C 的编译器生成。我们把这一系列的隐式规则叫作“隐式规则链”。

在上面的举例中，如果文件.c 存在，那么就直接调用 C 的编译器的隐式规则，如果没有.c 文件，但有一个.y 文件，那么 Yacc 的隐式规则会被调用，生成.c 文件，然后，再调用 C 编译的隐式规则，最终由.c 生成.o 文件，达到目标。

5.6.2 隐式规则中的变量

在隐式规则的命令中，基本上都是使用了一些预先设置的变量。可以在 Makefile 中改变这些变量的值，或是在 make 的命令行中传入这些值，或是在环境变量中设置这些值，无论怎么样，只要设置了这些特定的变量，那么它们就会对隐式规则起作用。当然，也可以利用 make 的“-R”或“--no－builtin-variables”参数来取消所定义的变量对隐式规则的作用。

例如，第一条隐式规则——编译 C 程序的隐式规则的命令是“$(CC) －c $(CFLAGS) $(CPPFLAGS)”。Make 默认的编译命令是“cc”，如果你把变量“$(CC)”重定义成“gcc”，把变量“$(CFLAGS)”重定义成“-g”，那么，隐式规则中的命令全部会以“gcc －c －g $(CPPFLAGS)”的样子来执行了。

可以把隐式规则中使用的变量分成两种：一种是命令相关的，如“CC”；一种是参数相关的，如“CFLAGS”。下面向大家介绍 Makefile 的隐式规则中常用的变量。表 5.4 列出了与命令相关的变量。

表 5.4 与命令相关的变量

变　量	含　义
AR	函数库打包程序，默认命令是“ar”
AS	汇编语言编译程序，默认命令是“as”

(续表)

变　量	含　义
CC	C 语言编译程序。默认命令是“cc”
CXX	C++语言编译程序。默认命令是“g++”
CO	从 RCS 文件中扩展文件程序。默认命令是“co”
CPP	C 程序的预处理器(输出是标准输出设备)。默认命令是“$(CC) – E”
FC	Fortran 和 Ratfor 的编译器和预处理程序。默认命令是“f 77”
GET	从 SCCS 文件中扩展文件的程序。默认命令是“get”
LEX	Lex 方法分析器程序(针对 C 或 Ratfor)。默认命令是“lex”
PC	Pascal 语言编译程序。默认命令是“pc”
YACC	Yacc 文法分析器(针对 C 程序)。默认命令是“yacc”
YACCR	Yacc 文法分析器(针对 Ratfor 程序)。默认命令是“yacc – r”
MAKEINFO	转换 Texinfo 源文件(.texi)到 Info 文件程序。默认命令是“makeinfo”
TEX	从 TeX 源文件创建 TeX DVI 文件的程序。默认命令是“tex”
WEAVE	转换 Web 到 TeX 的程序。默认命令是“weave”
TEXI2DVI	从 Texinfo 源文件创建 TeX DVI 文件的程序。默认命令是“texi2dvi”
CWEAVE	转换 C Web 到 TeX 的程序。默认命令是“cweave”
TANGLE	转换 Web 到 Pascal 语言的程序。默认命令是“tangle”
CTANGLE	转换 C Web 到 C。默认命令是“ctangle”
RM	删除文件命令。默认命令是“rm – f”

表 5.5 列出的是关于命令参数的变量。对于这些变量，如果没有指明其值，那么其默认值都是空。

表 5.5　与参数相关的变量

变　量	含　义
ARFLAGS	函数库打包程序 AR 命令的参数。默认值是“rv”
ASFLAGS	汇编语言编译器参数(当明显地调用“.s”或“.S”文件时)
CFLAGS	C 语言编译器参数
CXXFLAGS	C++语言编译器参数
COFLAGS	RCS 命令参数
CPPFLAGS	C 预处理器参数(C 和 Fortran 编译器也会用到)
FFLAGS	Fortran 语言编译器参数
GFLAGS	SCCS“get”程序参数
LDFLAGS	链接器参数(如：“ld”)
LFLAGS	Lex 文法分析器参数
PFLAGS	Pascal 语言编译器参数

(续表)

变　量	含　义
RFLAGS	Ratfor 程序的 Fortran 编译器参数
YFLAGS	Yacc 文法分析器参数

5.6.3　使用模式规则

可以使用模式规则来定义一个隐式规则。模式规则与一般的规则类似，只是在模式规则中，目标的定义需要有“%”字符。“%”定义对文件名的匹配，表示任意长度的非空字符串。在依赖目标中同样可以使用“%”，只是依赖目标中“%”的取值，取决于其目标。

注 意

模式规则中“%”的展开和变量与函数的展开是有区别的，“%”的展开发生在变量和函数的展开之后。变量和函数的展开发生在 make 载入 Makefile 时，而“%”的展开则发生在运行时。

1. 模式规则举例

模式规则中，至少在规则的目标中要包含“%”符号。例如：“%.c”表示以.c 结尾的文件名(文件名的长度至少为 3)，“s.%.c”表示以 s.开头，以.c 结尾的文件名(文件名的长度至少为 5)。如果“%”定义在目标中，那么目标中“%”的值决定了依赖目标中的“%”的值，也就是说，目标中的模式的“%”决定了依赖目标中“%”的样子。例如有一个模式规则如下：

```
%.o : %.c ; <command ......>
```

其含义是，指出了从所有的.c 文件生成相应的.o 文件的规则。如果要生成的目标是“a.o b.o”，那么“%c”就是“a.c b.c”。

一旦依赖目标中的“%”模式被确定，make 便会被要求去匹配当前目录下所有的文件名，一旦找到，make 就会执行规则下的命令，所以在模式规则中，目标可能会是多个的，如果有模式匹配出多个目标，make 就会产生所有的模式目标，此时，make 关心的是依赖的文件名和生成目标的命令这两件事。

下面这个例子表示把所有的.c 文件都编译成.o 文件：

```
%.o : %.c
$(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

其中，“$@”表示所有目标的集合，“$<”表示所有依赖目标的集合(在模式定义规则的情形下)。这些奇怪的变量被我们叫作自动化变量，稍后会向读者介绍。

下面的这个例子中有两个目标是模式的：

```
%.tab.c %.tab.h: %.y
bison -d $<
```

这条规则告诉 make 把所有的.y 文件都以 bison -d <n>.y 执行，然后生成<n>.tab.c 和<n>.tab.h 文件(<n>表示一个任意字符串)。如果执行程序 foo 依赖于文件 parse.tab.o 和 scan.o，并且文件 scan.o 依赖于文件 parse.tab.h，如果 parse.y 文件被更新了，那么根据上述的规则，bison -d parse.y

就会被执行一次，于是，parse.tab.o 和 scan.o 的依赖文件就齐了(假设，parse.tab.o 由 parse.tab.c 生成，scan.o 由 scan.c 生成，而 foo 由 parse.tab.o 和 scan.o 链接生成，而且 foo 和其.o 文件的依赖关系也已写好，那么，所有的目标都会得到满足)。

2. 自动化变量

在上述的模式规则中，目标和依赖文件都是一系列的文件，那么如何书写一个命令来完成从不同的依赖文件生成相应的目标？因为在每一次对模式规则的解析时，都会是不同的目标和依赖文件。

自动化变量就是完成这个功能的。在前面已经向读者多次提到了自动化变量。所谓自动化变量，就是这种变量会把模式中所定义的一系列的文件自动地逐个取出，直至所有的符合模式的文件都取完了。这种自动化变量只应出现在规则的命令中。表 5.6 是 Makefile 中所有的自动化变量及其含义。

表 5.6 Makefile 的自动化变量

变 量	含 义
$@	表示规则中的所有目标文件的集合。在模式规则中如果有多个目标，“$@”就是匹配于目标中模式定义的集合
$%	仅当目标是函数库文件时，表示规则中的目标成员名，如果目标不是函数库文件(UNIX 下是.a，Windows 下是.lib)，其值为空
$<	依赖目标中的第一个目标名字，如果依赖目标是以模式(即“%”)定义的，则“$<”是符合模式的一系列的文件集
$?	所有比目标新的依赖目标的集合，以空格分隔
$^	所有的依赖目标的集合，以空格分隔。如果在依赖目标中有多个重复的，则自动去除重复的依赖目标，只保留一份
$+	同“$^”，也是所有依赖目标的集合，只是它不去除重复的依赖目标
$*	目标模式中“%”及其之前的部分
$(@D)	“$@”的目录部分(不以斜杠作为结尾)，如果“$@”中没有包含斜杠，其值为“.”(当前目录)
$(@F)	“$@”的文件部分，相当于函数“$(notdir $@)”
$(*D)	同“$(@D)”，取文件的目录部分
$(*F)	同“$(@F)”，取文件部分，但不取后缀名
$(%D)	函数包文件成员的目录部分
$(%F)	函数包文件成员的文件名部分
$(<D)	依赖目标中的第一个目标的目录部分
$(<F)	依赖目标中的第一个目标的文件名部分
$(^D)	所有依赖目标文件中的目录部分(无相同的)
$(^F)	所有依赖目标文件中的文件名部分(无相同的)
$(+D)	所有依赖目标文件中的目录部分(可以有相同的)
$(+F)	所有依赖目标文件中的文件名部分(可以有相同的)

(续表)

变　量	含　义
$(?D)	所有被更新文件的目录部分
$(?F)	所有被更新文件的文件名部分

提 示

Makefile 中共有 21 个自动化变量，在表 5.6 中，后面的 14 个变量只是在前面 7 个变量后分别加上了字母“D”和“F”，字母“D”代表 Directory，就是目录，“F”代表 File，就是文件。

另外，对于变量“$<”，为了避免产生不必要的麻烦，最好为$后面的那个特定字符加上圆括号，比如“$(<)”就要比“$<”好一些。

3. 模式的匹配

在模式规则中，把“%”所匹配的内容叫作“茎”。例如，假设“%.c”所匹配的文件是“test.c”，那么其中的“test”就是“茎”。在目标和依赖目标中同时有“%”时，依赖目标的“茎”会传给目标，当作目标中的“茎”。

当一个模式匹配包含有斜杠(实际也不经常包含)的文件时，那么在进行模式匹配时，目录部分会首先被移开，然后再进行匹配，成功后再把目录加回去。在进行“茎”的传递时，我们需要知道这个步骤。例如有一个模式“e%t”，文件“src/eat”匹配于该模式，于是“src/a”就是其“茎”，如果这个模式定义在依赖目标中，而被依赖于这个模式的目标中又有个模式“c%r”，那么，目标就是“src/car”(“茎”被传递)。

5.7 本章小结

本章向读者介绍了 Linux 下的工程管理器 make 的运行机制，以及 Makefile 的书写规则，包括变量和函数的使用，隐式规则等。编写 Makefile，是在 Linux 下开发较大型的程序工程时所必需的，它大大提高了实际项目的工作效率。Makefile 的书写规则内容较多，而且比较容易出错，初学者可能不易熟练掌握，需要读者在实践中不断掌握和深化相关知识。

实战演练

1. 若在一个工程中包含了 4 个 C 源文件和 1 个头文件，它们分别是 main.c、fun1.c、fun2.c、fun3.c 和 myhead.h，试编写一个 Makefile 文件，能够使用 make 生成最终的可执行文件 all。

2. 使用通配符“*.o”表示当前目录下所有以后缀名为.o 的文件，试使用 rm 命令来删除当前目录下所有的.o 文件。

3. 编写一个 Makefile，其中定义一个伪目标 phonyall，其依赖于 4 个目标 prog1、prog2 和 prog3，当输入一个“make”命令时，使所有的目标都能够被编译。

4. 定义一个变量 var，并赋予其值为“Hello, Linux C program”，试使用 echo 命令来回显该变量的值。

5. 在 Makefile 中使用字符串替换函数 subst，将字符串“sweat will become to sweet”中的字符“e”替换成大写字母“E”。

6. 在 Makefile 中使用取单词串函数 wordlist，取出字符串“I like Linux C programs”中的第 2～8 个单词。

7. 在 Makefile 中使用取文件名函数 notdir，返回下列各路径的文件名：/root/install.log、/home/zhangfan/hello.txt、hello.c。

8. 在 Makefile 中使用取后缀名函数 suffix，返回下列各文件的后缀名：hello.c、foo.s、fun.o 及 usr/src/linux-2.4/Makefile。

第 II 部分

提 高 篇



第 6 章

文件I/O操作

文件操作是Linux系统中最常见的操作之一，本章介绍Linux系统中的文件及和文件有关的I/O操作。在Linux中，有关I/O的操作可以分为两类：基于文件描述符的I/O操作和基于流的I/O操作，它们有各自不同的特点和优势，有些情况下它们是可以相互替代的，而有些情况则是不可以相互替代的。本章向读者介绍基于文件描述符的I/O操作，基于流的I/O操作将在下一章进行介绍。



本章内容：

◎ Linux文件系统简介。
◎ 基于文件描述符的I/O操作。
◎ 文件的属性操作。
◎ 文件的其他操作。
◎ 特殊文件的操作。

6.1 软件编程体系简介

文件是 Linux 环境中一个相当重要的概念，文件提供了简单并一致的接口来处理系统服务与设备。在 Linux 中，一切都是文件。也就是说，在 Linux 中，所有的内容都被看成文件，所有的操作都可以归结为对文件的操作，操作系统可以像处理普通文件一样来使用磁盘文件、串口、键盘、显示器、打印机及其他的设备。

6.1.1 Linux 的文件系统结构

文件结构是文件存放在磁盘等存储设备中的组织方法，主要体现在对文件和目录的组织上。目录提供了管理文件的一个方便而有效的途径，用户能够从一个目录切换到另一个目录，而且可以设置目录和文件的权限，设置文件的共享程度。

Linux 文件系统是目录和文件的一种层次安排，目录的起点称为根(root)，其名字是一个字符“/”。目录(directory)是一个包含目录项的文件，在逻辑上，可以认为每个目录项都包含一个文件名，同时还包含说明该文件属性的信息。文件属性是文件类型、文件长度、文件所有者、文件的许可权(例如，其他用户是否能访问该文件)、文件最后的修改时间等。

使用 Linux，用户可以设置目录和文件的权限，以便允许或拒绝其他人对其进行访问。Linux 目录采用多级树形结构，图 6.1 表示了根目录下包含的目录。通过这种树形等级结构，用户可以浏览整个系统，可以进入任何一个已授权进入的目录，访问那里的文件。

文件结构的相互关联性使共享数据变得很容易，几个用户可以访问同一个文件。Linux 是一个多用户系统，操作系统本身的驻留程序存放在以根目录开始的专用目录中，有时被指定为系统目录。图 6.1 中那些根目录下的目录就是系统目录。

内核、Shell 和文件结构一起形成了 Linux 的基本操作系统结构。它们使得用户可以运行程序，管理文件及使用系统。此外，Linux 操作系统还有许多被称为实用工具的程序，辅助用户完成一些特定的任务。

提 示

在使用 Linux 的文件及目录时，用户可遵循以下的技巧：

- 用户文件存放在/home/user_login_name 目录下(及其子目录下)。
- 本地管理员在大多数情况下将额外的软件安装在/usr/local 目录下，并将符号连接在/usr/local/bin 下的主执行程序。
- 系统的所有设置均在/etc 目录下。
- 不要修改根目录(“/”)或/usr 目录下的任何内容，除非真的清楚要做什么。这些目录最好和 Linux 发布时保持一致。
- 大多数的系统工具和应用程序安装在目录/bin、/usr/sbin、/sbin、/usr/x11/bin 及/usr/local/bin 下。
- 系统中所有的文件均在单一的目录树下。
- Linux 没有所谓的“驱动符”，所有的外围设备都当作文件来处理，即设备文件。

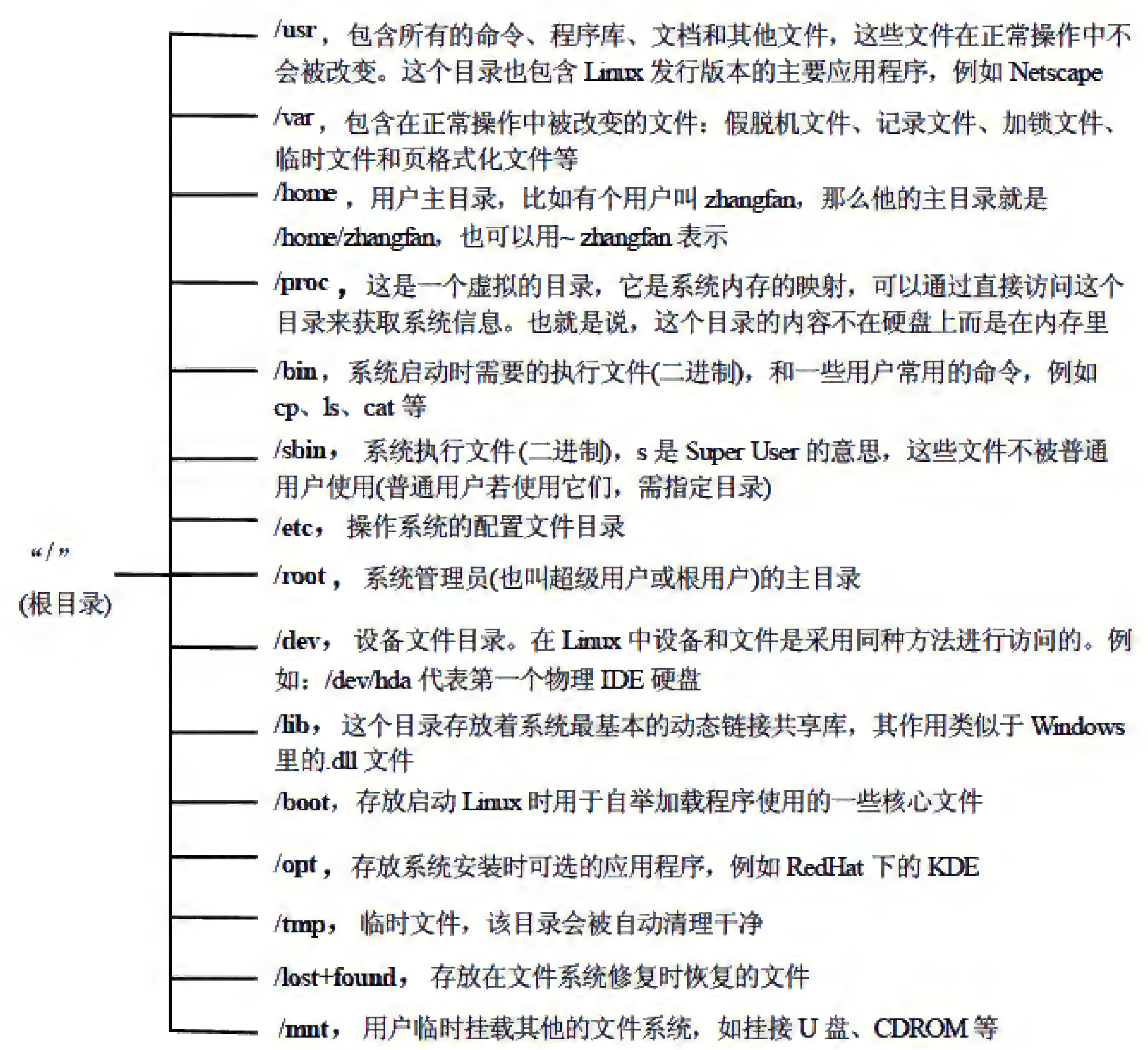


图 6.1　Linux 根目录下包含的目录

6.1.2　文件类型

先来看一个例子，在终端输入“ls -l”命令，系统会打印出当前目录下所有文件的信息：

```
# ls -l
-rw-r--r--    1 root root       2   03-27 02:00 fonts.scale
-rw-r--r--    1 root root      53K 03-16 08:54 install.log
drwxr-xr-x 2 1000 users    4.0K 04-04 23:30 mkuml-2004.07.17
drwxr-xr-x 2 root root     4.0K 04-19 10:53 mydir
```

这个例子是 Linux 用户再熟悉不过的了，它打印出了当前目录下所有文件的信息，包括文件类型、文件属性、用户名、用户所在组、文件大小、修改时间、文件名等。而其中的第一栏信息(文件类型和文件属性)就是本小节和下一小节将要向读者介绍的内容，也是关于 Linux 文件的重要内容。

第一栏的信息包含了 10 位字符，分为 4 组，如图 6.2 所示。第 1 组即第 1 位，表示文件的类型；第 2 组为 2-4 位，代表文件所有者(User)的权限，分别为读、写、执行；第 3 组为 5-7 位，代表文件所有者的同组用户(Group)的权限，分别为读、写、执行；第 4 组为 8-10 位，代表其他组用户(Other)权限，同样分别为读、写、执行。

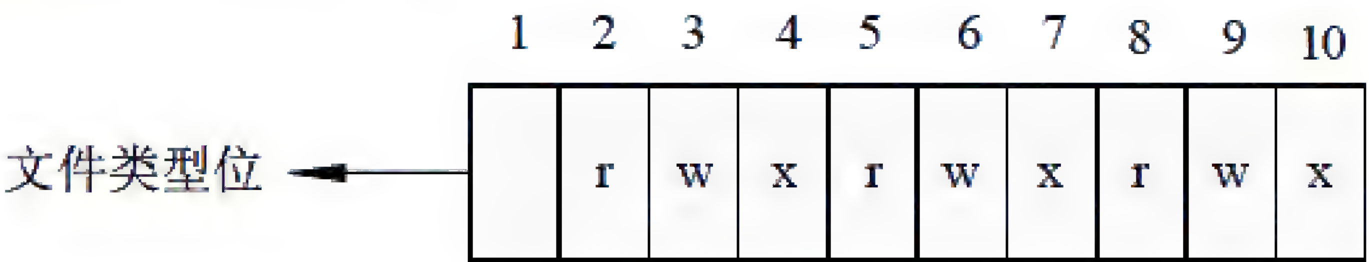


图 6.2　文件属性位含义

在图 6.2 中，第一个字符(比如“-”和“d”)显示了该文件的类型，不同字符表示了不同的文件类型，如表 6.1 所示。

表 6.1　Linux 文件类型符号

符　号	文 件 类 型	符　号	文 件 类 型
-	普通文件	d	目录文件
l	链接文件	b	块设备文件
c	字符设备文件	p	管道文件
s	套接口文件		

下面给出它们的具体介绍。

1. 普通文件

普通文件是计算机用户和操作系统用于存放数据、程序等信息的文件，一般都长期地存放在外存储器(磁盘等)中。普通文件一般又分为文本文件和二进制文件。

(1) 文本文件：这是 Linux 系统中最多的一种文件类型，之所以称为文本文件，是因为它的内容是用户可以直接读到的数据，例如数字、字母等。系统设置文件几乎都属于这种文件类型。举例来说，使用命令“cat hello.c”就可以看到当前目录下的 hello.c 文件的内容——前提是这个文件存在(cat 命令是查看某个文件的内容)。

(2) 二进制文件：操作系统事实上仅认识且可以执行二进制文件(binary file)，而用户通常是不能直接查看它的内容的。Linux 中的可执行文件(脚本、文本方式的批处理文件不算)就是这种格式的。举例来说，命令 cat 本身就是一个二进制文件。

2. 目录文件

目录文件是文件系统中一个目录所包含的目录项组成的文件，目录文件只允许系统进行修改，用户进程可以读取目录文件，但不能对它们进行修改。也就是说，对一个目录文件具有读许可权的任一进程都可以读该目录的内容，但只有内核才可以写目录文件。

3. 设备文件

设备文件是用于为操作系统与 I/O 设备提供连接的一种文件，分为字符设备文件和块设备文件，对应于字符设备和块设备。Linux 把对设备的 I/O 作为普通文件的读取/写入，操作内核提供了对设备处理和对文件处理的统一接口。每一种 I/O 设备对应一个设备文件，存放在/dev 目录中，如行式打印机对应于文件/dev/lp。目前在最新的 Linux 发行版本中，一般不用用户创建设备文件，因为这些文件是和内核相关联的。

在设备文件中有一个极其特殊的文件/dev/null。所有放入这一设备的数据都将不存在，可以将它看作是删除操作。

说明

字符设备与块设备：

- 字符设备(Character device)：这是一个顺序的数据流设备，对这种设备的读写是按字符进行的，而且这些字符是连续地形成一个数据流。字符设备不具备缓冲区，所以对这种设备的读写是实时的，如串口终端、磁带机等。
- 块设备(block device)：这是一种具有一定结构的随机存取设备，对这种设备的读写是按块进行的，它使用缓冲区来存放暂时的数据，待条件成熟后，从缓存一次性写入设备或从设备中一次性读出放入到缓冲区，如磁盘和文件系统等。

下面的例子显示了/dev/tty(串口)和/dev/hda1(硬盘)文件的信息：

```
# ls -l /dev/tty
crw-rw-rw- 1 root tty 5,    0 04-19 08:29    /dev/tty
# ls -l /dev/hda1
brw-r-----    1 root disk 3, 1 2006-04-19    /dev/hda1
```

从中可以看出，/dev/tty 的文件类型显示为字符“c”，/dev/hda1 文件类型显示为字符“b”。

4. 链接文件

链接文件又称符号链接文件，它提供了共享文件的一种方法，在链接文件中不是通过文件名实现文件共享，而是通过链接文件所包含的指向文件的指针来实现对文件的访问。普通用户可以建立链接文件，并通过其指针访问它所指向的那个文件。使用链接文件可以访问普通文件，还可以访问目录文件和不具有普通文件实态的其他文件，也就是说，链接文件可以在不同的文件系统之间建立一种链接关系。根据链接对象的不同，链接文件又可以分为硬链接文件和符号链接文件。

5. 管道文件

管道文件主要用于在进程间传递数据，它是 Linux 进程间的一种通信机制(在第 10 章将会讲到)。管道是进程间传递数据的“媒介”，某个进程将数据写入管道的一端，另一个进程从管道另一端读取数据。Linux 对管道的操作与文件操作相同，把管道作为文件进行处理，管道文件又可以分为匿名管道和命名管道两种。

6. 套接口文件

套接口(Socket)文件主要用于在不同计算机的进程间的通信，也称为套接字。

套接口是操作系统内核中的一个数据结构，它是网络中的节点进行相互通信的门户。套接口有 3 种类型：流式套接口、数据报套接口和原始套接口。流式套接口也就是 TCP 套接口(或称面向连接的套接口)，数据报套接口也就是 UDP 套接口(或称无连接的套接口)，原始套接口用“SOCK_RAW”表示。

流式套接口定义了一种可靠的面向连接的服务，实现了无差错无重复的顺序数据传输。数据报套接口定义了一种无连接的服务，数据通过相互独立的报文进行传输，是无序的，并且不保证可靠、无差错。原始套接口允许对低层协议如 IP 或 ICMP 直接访问，主要用于新的网络协议实现的测试等。

在本书的第 11 章中将详细讲述套接口，以及基于套接口的网络编程。

6.1.3 文件访问权限

所谓权限，指的是文件系统为了进行安全管理需要在对文件操作时进行的用户身份认证。合法的用户可以进行操作，而没有权限的用户就不能对文件进行操作。

Linux 系统是一个典型的多用户操作系统，不同的用户处于不同的地位。为了保护系统的安全性，Linux 系统对不同用户访问同一文件的权限做了不同的规定，即不同的用户具有不同的读、写和执行的权限。读者在 6.1.2 小节的几个例子中已经看到了文件的权限符号：r(读)、w(写)和 x(执行)。

用户在登录 Linux 时，系统自动为其分配一个“身份证号”，来区分不同用户的权限级别，称为 ID 号。

对于一个文件来说，它有一个特定的所有者，也就是对文件具有所有权的用户。同时，由于在 Linux 系统中，用户是按组分类的，一个用户属于一个或多个组，所以文件所有者以外的用户又可以分为所有者的同组用户和其他组用户。因此 Linux 系统按文件所有者、文件所有者同组用户和其他组用户这 3 类规定不同的文件访问权限。

提 示

系统管理员 root 用户(也称为根用户，或超级用户)是一个非常特别的用户，此用户对系统具有最高的控制权。对于系统中的所有文件，root 用户都有读、写和执行的权限(当然，前提是该文件是可读、可写或者可执行的)。

还是 6.1.2 小节开始的那个例子：

```
# ls -l
-rw-r--r--    1 root root        2   03-27 02:00 fonts.scale
-rw-r--r--    1 root root       53K 03-16 08:54 install.log
drwxr-xr-x 2 1000 users    4.0K 04-04 23:30 mkuml-2004.07.17
drwxr-xr-x 2 root root     4.0K 04-19 10:53 mydir
```

对于 fonts.scale 文件来讲，“rw-r--r-”字串说明了它的属性：用户可读可写，用户所在组(同组用户)可读，其他组用户可读。

用样，对于 mydir 文件来讲，“rwxr-xr-x”字串说明了它的属性：用户可读可写可执行，用户所在组(同组用户)可读可执行，其他组用户可读可执行。

6.2 基于文件描述符的 I/O 操作

在本节中读者将看到在 Linux 下如何创建文件、打开文件、读取文件、写入文件及关闭文件等。这些函数经常被称之为不带缓存的 I/O(unbuffered I/O，与将在第 7 章中说明的基于流的 I/O 函数相对照)。这些不带缓存的 I/O 函数不是 ANSI C 的组成部分，但却是 POSIX.1 和 XPG 3 的组成部分。

6.2.1 文件描述符

Linux 操作系统内核(kernel)利用文件描述符(file descriptor)来访问文件。文件描述符是一个非负整数，是一个用于描述被打开文件的索引值，它指向该文件的相关信息记录表。当内核打开一个现存文件或创建一个新文件时，就会返回一个文件描述符。当读、写文件时，也需要使用文件描述符来指定待读写的文件。

按照惯例，UNIX shell 将文件描述符 0 与进程(关于进程的概念，将在本书的第 8 章介绍)的标准输入(standard input)相结合，文件描述符 1 与标准输出(standard output)相结合，文件描述符 2 与标准出错(standard error)相结合。尽管这种习惯并非 UNIX 内核的特性，但是因为一些 Shell 和很多应用程序都使用这种习惯，因此，如果内核不遵循这种习惯的话，很多应用程序将不能使用。

POSIX(Portable Operating System Interface，可移植操作系统接口，缩写为 POSIX 是为了读音更像 UNIX)定义了 STDIN_FILENO、STDOUT_FILENO 和 STDERR_FILENO 来代替 0、1、2。这 3 个符号常量的定义位于头文件 unistd.h。

文件描述符的有效范围是 0 到 OPEN_MAX。一般来说，每个进程最多可以打开 1024 个文件(0～1023)，这个值可以使用 ulimit -n 命令来查看。文件描述符是由无符号整数表示的句柄，进程使用它来标识打开的文件。文件描述符与包括相关信息(如文件的打开模式、文件的位置类型、文件的初始类型等)的文件对象相关联，这些信息被称作文件的上下文。

6.2.2 标准输入、标准输出和标准出错

在 UNIX/Linux 系统中，每当运行一个新程序时，所有的 Shell 都为其打开 3 个文件描述符：标准输入、标准输出及标准出错。通俗地讲，这里的标准输入、标准输出及标准出错就是 3 个不同的文件描述符(在下一章读者还将会看到基于流的标准输入、标准输出及标准出错)，它们对应的值分别为 0、1、2。表 6.2 列出了三者各自的对应关系。

表 6.2 标准输入、标准输出及标准出错

类　型	文件描述符	说　明
标准输入	0	它是命令的输入，默认是键盘，也可以是文件或其他命令的输出
标准输出	1	它是命令的输出，默认是屏幕，也可以是文件
标准出错	2	它是命令出错信息的输出，默认是屏幕，也可以是文件

细心的读者会发现，在表 6.2 中，标准输入、输出及出错除了可以是默认的键盘或显示器外，还可以是文件或其他命令的输出，这就是下面将要介绍的文件重定向的概念。

6.2.3 文件重定向

大多数 Shell 都提供一种方法，使任何一个或所有这 3 个描述符都能重新定向到某一个文件，而不是用默认的输入或输出设备。例如：

```
ls > file.list
```

执行 ls 命令，其标准输出重新定向到名为 file.list 的文件上。那么此时在屏幕终端上就不会显示 ls 命令的执行结果，它的执行结果在 file.list 文件中，需要注意的是即使 file.list 文件不存在，重定向命令“>”也会创建这个文件。

1. 重定向标准输出

重定向标准输出，即不使用系统标准输出的默认设备，而将输出结果直接写在一个新的文件中。命令格式如下：

```
command > filename      #把标准输出重定向到 filename 文件中
command >> filename     #把标准输出重定向到 filename 文件中，方式是追加在现有内容的后面
command 1> filename     #把标准输出重定向到一个文件中
> myfile                #创建一个长度为 0 的空文件
```

说 明

command 代表用户所熟悉的 shell 命令，filename 表示文件名，“#”后为笔者对该命令的注释。另外，用户使用时注意在重定向符号“<”、“<<”、“>”或“>>”的前后都有一个空格，否则 Shell 会把它们都当作是输入的命令，而不是重定向符号，而导致错误。

值得一提的是，上面的第三条命令“command 1> filename”中使用了标准输出的文件描述符 1，用它来区别是标准输出，还是标准出错。当然，1 也可以省略不写，如第一条命令“command > filename”。另外需要注意，文件描述符 1(或 2)与前面的命令之间有空格，而与标准输出(或标准出错)的重定向符“>”、“>>”之间没有空格，否则也将导致错误。

2. 重定向标准输入

重定向标准输入，即不使用系统标准输入的默认设备，而是引用其他文件的内容或是其他命令的输出。命令格式如下：

```
command < filename      #以 filename 文件的内容作为 command 命令的标准输入
command < file1 > file2 #以 file1 文件的内容作为 command 命令的标准输入，并以 file2 文件作为命令
                         执行结果的标准输出
command << delimiter    #从标准输入中读入，直至遇到 delimiter 分界符
```

标准输入的文件描述符为 0，通常省略不写。

例如，我们通过执行下面一系列的命令，读者不难体会出重定向标准输出和重定向标准输入的含义(为便于读者理解，我们将命令行给出了详细的注释)：

```
# cat hello          #查看当前目录下 hello 文件中的内容
Hello! I like Linux C program!
I am doing Linux C programs!
# cat tmp            #查看当前目录下 tmp 文件中的内容，可见该文件内容为空
# cat < hello > tmp
                     #将 hello 文件作为 cat 命令的标准输入，并将 cat 的执行结果重定向到 tmp 文件中去
# cat hello          #再次查看 hello 文件中的内容，没有变化
Hello! I like Linux C program!
I am doing Linux C programs!
# cat tmp            #再次查看 tmp 文件中的内容，它是 cat < hello 命令的执行结果
```

```
Hello! I like Linux C program!
I am doing Linux C programs!
```

第一次查看 tmp 文件中的内容时，看到该文件的内容为空，执行完“cat < hello > tmp”命令后，再次查看 tmp 文件中的内容，发现它与 hello 文件中的内容是一致的了。

3. 重定向标准出错

将系统执行的错误信息重定向到一个文件中，而不使用默认的输出设备，如显示器等。命令格式如下：

```
command 2> filename     #把标准出错信息重定向到 filename 文件中
command 2>> filename    #把标准出错信息重定向到一个文件中(追加)
```

标准输出和标准出错可以结合使用，命令格式如下：

```
command 1> file1 2> file2     #将标准输出重定向到 file1 中，然后再把标准出错重定向到 file2 中
command > filename 2> &1    #把标准输出和标准出错一起重定向到 filename 文件中
command >> filename 2> &1 #把标准输出和标准出错一起重定向到 filename 文件中(追加)
command > filename 2> &1 << delimiter #把标准输出和标准出错一起重定向到 filename 文件中，直至遇到
                              delimiter 分界符
```

标准出错的文件描述符为 2，注意它是不可以省略的，否则就成为标准输出了。

读者可以从下面一系列的 shell 命令中体会出重定向标准出错的含义：

```
# xxx        #输入一个错误的命令，Shell 会产生标准出错信息，并在默认的输出设备中输出
bash: xxx: command not found
# cat tmp     #重定向之前先来查看当前目录下 tmp 文件中的内容
Hello! I like Linux C program!
I am doing Linux C programs!
# xxx 2>> tmp     #以追加的方式将标准出错重定向到 tmp 文件
# cat tmp          #再次查看 tmp 文件中的内容
Hello! I like Linux C program!
I am doing Linux C programs!
bash: xxx: command not found
```

可以看到，重定向标准出错之后，系统产生的错误信息不在默认的输出设备(显示器)上显示了，而是被重定向到了 tmp 文件中。

6.2.4 文件的创建、打开与关闭

要对一个文件进行操作，首先要求这个文件存在，其次是要在操作之前将这个文件打开。这样才能实现对该文件的操作，当完成操作以后，则必须将文件关闭。文件的创建、打开与关闭是文件 I/O 操作的第一步。

1. open 函数

调用 open 函数可以打开或创建一个文件。函数说明如下：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open (const char *pathname, int flags ) ;                    /*打开一个现有的文件*/
int open (const char *pathname, int flags, mode_t mode ) ; /*打开的文件不存在，则先创建它*/
```

返回：若成功，返回文件描述符，若出错为-1。

其中参数 pathname 是一个字符串指针，它指向需要打开(或创建)文件的绝对路径名或相对路径名。

说明

在 Linux 中，0 个或多个以斜线分隔的文件名(目录名)序列构成路径名(pathname)，其中以根目录“/”开始的路径名称为绝对路径名(absolute pathname)，如“/home/zhangfan/hello.c”，否则称为相对路径名(relative pathname)，如“hello.c”。

参数 flags 是用于描述文件打开方式的参数，它的具体取值及其含义如表 6.3 所示，这些常数定义在<fcntl.h>头文件中。具体调用时，flags 的值可由表中取值逻辑或得到。其中，O_RDONLY、O_WRONLY、O_RDWR 这 3 个参数应当只指定其中一个，其他常数则是可选择的。

表 6.3　flags 的取值及其含义

flags 取值	含　义
O_RDONLY	以只读方式打开文件
O_WRONLY	以只写方式打开文件
O_RDWR	以读写方式打开文件
O_CREAT	若所打开文件不存在则创建此文件。使用此选择项时，需同时使用第三个参数 mode 说明该新文件的存取许可权位
O_EXCL	如果同时指定了 O_CREAT，而文件已经存在，则导致调用出错
O_TRUNC	如果文件存在，而且为只读或只写方式打开，则将其长度截短为 0
O_NOCTTY	如果 pathname 指的是终端设备(tty)，则不将此设备分配作为此进程的控制终端
O_APPEND	每次写时都加到文件的尾端
O_NONBLOCK	如果 pathname 指的是一个 FIFO、一个块特殊文件或一个字符特殊文件，则此选择项为此文件的本次打开操作和后续的 I/O 操作设置为非阻塞方式
O_NONELAY	功能不完善的 O_NONBLOCK，通常不使用
O_SYNC	只在数据被写入外存或其他设备之后操作才返回

而参数 mode 用于指定所创建文件的权限，其取值及含义如表 6.4 所示。具体应用时，使用按位逻辑或的方法根据需要对其进行组合。

表 6.4 mode 取值及其含义

mode 取值	对应八进制数	含 义
S_ISUID	04000	设置用户识别号
S_ISGID	02000	设置组识别号
S_SVTX	01000	粘贴位
S_IRUSR	00400	文件所有者的读权限位
S_IWUSR	00200	文件所有者的写权限位
S_IXUSR	00100	文件所有者的执行权限位
S_IRGRP	00040	所有者同组用户的读权限位
S_IWGRP	00020	所有者同组用户的写权限位
S_IXGRP	00010	所有者同组用户的执行权限位
S_IROTH	00004	其他组用户的读权限位
S_IWOTH	00002	其他组用户的写权限位
S_IXOTH	00001	其他组用户的执行权限位

此外，为方便程序中的使用，<fcntl.h>头文件中还定义了 3 个常用的逻辑组合：

- S_IRWXU = S_IRUSR | S_IWUSR | S_IXUSR，文件所有者的读、写、执行权限，它是各个相应权限位的逻辑和。
- S_IRWXG = S_IRGRP | S_IWGRP | S_IXGRP，文件所有者同组用户的读、写、执行权限，它是各个相应权限位的逻辑和。
- S_IRWXO = S_IROTH | S_IWOTH | S_IXOTH，其他组用户的读、写、执行权限，它是各个相应权限位的逻辑和。

下面的例子说明了 open 函数的使用方法，它以只写方式打开(文件不存在则创建)一个文件，并将文件长度截短为 0，文件名由用户从终端输入。代码如程序 6.1 所示。

【程序 6.1】使用 open 函数打开(或创建)一个文件：open_file.c。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define FLAGS O_WRONLY | O_CREAT | O_TRUNC
/*定义 flags：只写，文件不存在则创建，文件长度截短为 0*/
#define MODE S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH
/*创建文件的权限：用户读、写、执行，组读、执行，其他用户读、执行*/
int main(void)
{
    const char *pathname;       /*指向需要打开(或创建)文件的绝对路径名或相对路径名*/
    int fd;                     /*文件描述符*/
```

```
        char pn[30];                    /*pn 为字符串数组，存放要打开(或创建)的文件名*/
        printf("Input the pathname[<30 strings]:");        /*输入路径名，小于 30 个字符*/
        gets(pn);
        pathname=pn;
        if((fd=open(pathname,FLAGS,MODE))==-1)    /*调用 open 函数*/
          {
              printf("error,open file failed!\n");
              exit(1);      /*出错退出*/
          }
        printf("OK,open file successful!\n");
        printf("fd=%d\n",fd);
        return 0;
    }
```

必须要向读者说明的是，这段代码是不健壮的，比如当用户从终端输入文件路径名时，只是简单地提示文件名应小于 30 个字符(通常来说 Linux 可以支持最大程度为 255 个字符的文件名)，但并没有进行进一步的判断，如果输入的文件名(路径名)大于 30 个字符，字符串数组 pn 就会产生溢出，程序无法执行。当然，这里只是给读者一个使用 open 函数的例子，我们的重点不是在判断终端的输入上。

使用本书在第 4 章已经向读者介绍的 gcc 编译器编译程序 6.1，并生成可执行文件 open_file，执行情况如下：

```
#gcc –o open_file open_file.c
warning: In function `main':
: the `gets' function is dangerous and should not be used.
#./open_file
Input the pathname[<30 strings]: /home/zhangfan/CODE/hello  ↙(↙表示回车)
OK,open file successful!
fd=3
```

从中可以看到，系统成功打开(或创建了)/home/zhangfan/CODE/hello 文件，文件描述符为 3。使用 ls –l 进一步查看该文件的信息：

```
# ls –l /home/zhangfan/CODE/hello
-rwxr-xr-x      1 root       root                0    9 月   19 21:43 hello
```

由此可见，文件的执行权限“rwxr-xr-x”与程序中的“MODE”宏定义是吻合的，文件的长度为 0，这与“FLAGS”宏定义是吻合的。

说 明

正如读者所看到的，在编译 code_6.1.c 时产生了警告信息，这是因为程序中 gets 函数的使用，读者将在第 7 章看到 gets 的缺陷和漏洞。尽管如此，在本章的一些程序代码中，我们仍是使用 gets 函数，先把警告信息放在一边。

另外要提醒读者的一点是，由 open 函数返回的文件描述符一定是最小的未用描述符数字。这一点被很多应用程序用来在标准输入、标准输出或标准出错上打开一个新的文件。例如，一个应用程序可以先关闭标准输出(通常是文件描述符 1)，然后打开另一个文件，事先就能了解

该文件一定会在文件描述符 1 上打开。

2. creat 函数

创建文件使用 creat 函数，函数原型如下：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat (const char *pathname, mode_t mode) ;
```

返回：若成功，返回以只写方式打开的文件描述符，若出错为-1。

参数 pathname 和 mode 的含义与 open 函数相同。

注意，creat 函数等效于：

```
open (pathname, O_WRONLY | OCREAT | O_TRUNC, mode) ;
```

creat 的一个不足之处是它以只写方式打开所创建的文件。在提供 open 的新版本之前，如果要创建一个临时文件，并要先写该文件，然后又读该文件，则必须先调用 creat、close，然后再调用 open。现在则可用下列方式调用 open：

```
open (pathname, O_RDWR | O_CREAT | O_TRUNC, mode) ;
```

3. close 函数

close 函数用于关闭一个文件，函数说明如下：

```
#include <unistd.h>
int close ( int fd );
```

返回：若成功返回 0，若出错返回-1。

参数 fd 是需关闭文件的文件描述符。

当对文件进行打开和关闭操作时，还会对其相关信息产生相应的影响。当打开一个文件时，该文件描述中的引用计数器值加 1。而关闭一个文件时，该文件描述中的引用计数器值减 1。当引用计数器的值减为 0 时，系统调用 close 不仅将释放该文件的描述符，而且也将释放该文件所占的描述表项。

关闭一个文件时也释放该进程加在该文件上的所有记录锁。当一个进程终止时，它所有的打开文件都由内核自动关闭。很多程序都使用这一功能而不显式地用 close 关闭打开的文件。

另外，当关闭的不是一个普通文件时，可能会产生一些其他的影响。例如，关闭管道文件的一端时，将影响到管道的另一端。

6.2.5 文件的定位

每个已打开的文件都有一个与其相关联的“当前文件位移量”，它是一个非负整数，用以度量从文件开始处计算的字节数。通常，读、写操作都从当前文件位移量处开始，并使位移量增加所读或写的字节数。按系统默认设置，当打开一个文件时，除非指定 O_APPEND 选择项，否则该位移量被设置为 0。

可以调用 lseek 函数显式地定位一个打开文件，函数说明如下：

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek (int fd, off_t offset, int whence) ;
```

返回：若成功，返回新的文件位移量，若出错为-1。

提 示

“lseek” 中的字符 “l” 表示长整型。

参数 fd 表示已打开文件的描述符，参数 offset 表示位移量的大小，单位字节，对参数 offset 的解释与参数 whence 的取值有关，如表 6.5 所示。

表 6.5 whence 取值及含义

whence 取值	含 义
SEEK_SET	将该文件的位移量设置为距文件开始处 offset 个字节
SEEK_CUR	将该文件的位移量设置为其当前值加 offset 个字节，offset 可为正或负
SEEK_END	将该文件的位移量设置为文件长度加 offset 个字节，offset 可为正或负

从表 6.5 中可以看到，当 whence 的取值为 SEEK_CUR 或 SEEK_END 时，参数 offset 允许取负值，这表示在当前位置或文件末尾位置上向前移动 offset 个字节。

另外，由于 lseek 成功执行时返回新的文件位移量，为此可以用下列方式确定一个打开文件的当前位移量：

```
off_t currpos;                        /*定义变量 currpos 的数据类型为 off_t */
currpos = lseek (fd, 0, SEEK_CUR);    /*offset 值为 0*/
```

这种方法也可用来确定所涉及的文件是否可以设置位移量。如果文件描述符引用的是一个管道或 FIFO，则 lseek 返回-1，并将 errno 设置为 EPIPE。

程序 6.2 用于测试其标准输入能否被设置位移量。

【程序 6.2】测试标准输入能否被设置位移量：offset_test.c。

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void)
{
    if (lseek (0, 0, SEEK_CUR) = -1)      /*判断标准输入文件能否被设置位移量*/
        printf("cannot seek!\n");
    else
        printf("seek OK!\n");
    exit(0);
}
```

使用 gcc 编译 offset_test.c，并生成可执行文件 offset_test：

```
#gcc –o offset_test offset_test.c
```

运行程序，得到输出结果：

```
# ./offset_test                              /*判断标准输入能否被设置位移量*/
cannot seek!
# ./offset_test < /etc/motd                  /*判断/etc/motd 文件能否被设置位移量*/
seek OK!
# ./offset_test < /home/zhangfan/CODE/hello  /*判断 hello 文件能否被设置位移量*/
seek OK!
```

说 明

在上面的这段命令行中，第二条命令使用了本章中讲述的重定向，即以/etc/motd 文件作为./offset_test 命令的标准输入，而第三条命令是以/home/zhangfan/CODE/hello 文件作为./offset_test 命令的标准输入。而对于第一条命令，没有重定向时，则采用默认的标准输入设备(键盘)。

从程序的运行结果可以看出，对于标准输入，一般不能设置位移量，而对于系统中的文件(如/etc/motd)，以及用户创建的一般文件(如我们前面创建的 hello 文件)是可以设置位移量的。

通常情况下，文件的当前位移量应当是一个非负整数，但是，某些设备也可能允许负的位移量。但对于普通文件，则其位移量必须是非负值。因为位移量可能是负值，所以在比较 lseek 的返回值时应当谨慎，不要测试它是否小于 0，而要测试它是否等于-1。

lseek 仅将当前的文件位移量记录在内核内，它并不引起任何 I/O 操作。然后，该位移量用于下一个读或写操作。

文件位移量可以大于文件的当前长度，在这种情况下，对该文件的下一次写操作将延长该文件，并在文件中构成一个空洞，这一点是允许的。位于文件中但没有写过的字节都被读为 0。读者将会在下一小节看到一个这样的例子。

6.2.6 文件的读写

读写是文件 I/O 操作的核心内容。上面已经介绍了文件的创建、打开、关闭和定位，但是要实现文件的 I/O 操作就必须对其进行读写。文件的读写操作的系统调用分别是 read 和 write 函数，它们的详细说明如下。

1. read 函数

用 read 函数从打开文件中读取数据。函数说明如下：

```
#include <unistd.h>
ssize_t read (int fd, void *buf, size_t count) ;
```

返回：读到的字节数，若已到文件尾返回 0，若出错为-1。

其中参数 fd 表示要进行读操作的文件的描述符，buf 是一个指向缓冲区的指针，该缓冲区存放将要读取到终端的数据，count 表示本次操作将要读取的数据的字节数。

读操作从文件的当前位移量处开始，在成功返回之前，该位移量增加实际读得的字节数。有多种情况可使实际读到的字节数少于要求读的字节数：

- 读普通文件时，在读到要求字节数之前已到达了文件尾端。例如，若在到达文件尾端之前还有 30 个字节，而要求读 100 个字节，则 read 返回 30，下一次再调用 read 时，它将返回 0(文件尾端)。
- 当从终端设备读时，通常一次最多读一行。
- 当从网络中读时，网络中的缓冲机构可能造成返回值小于所要求读的字节数。
- 某些面向记录的设备，例如磁带，一次最多返回一个记录。

2. write 函数

用 write 函数向打开文件写数据。函数说明如下：

```
#include <unistd.h>
ssize_t write (int fd, void *buf, size_t count );
```

返回：若成功为已写的字节数，若出错为-1。

其中参数 fd 表示要进行写操作的文件的描述符，buf 是一个指向缓冲区的指针，该缓冲区存放将要写入文件的数据，count 表示本次操作将要写入文件的数据的字节数。

函数返回值通常与参数 count 的值相同，否则表示出错。write 出错的一个常见原因是磁盘已写满，或者超过了对一个给定进程的文件长度限制。

对于普通文件，写操作从文件的当前位移量处开始。如果在打开该文件时，指定了 O_APPEND 选择项，则在每次写操作之前，将文件位移量设置在文件的当前结尾处。在一次成功操作写之后，该文件位移量增加实际写的字节数。

下面是一个写文件的例子，代码如程序 6.3 所示。假如在/home/zhangfan 目录下存在一个文本文件 hello，程序 6.3 将在该文件后追加写入用户从终端输入的字符(字符长度设为小于 80)。

【程序 6.3】使用 write 函数向文件写入数据：write_file.c。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define FILENAME "/home/zhangfan/hello"         /*要进行写操作的文件*/
#define SIZE 80                                 /*定义缓冲区大小*/
#define FLAGS O_RDWR | O_APPEND
/*定义参数 flags：以读写方式打开文件，向文件添加内容时从文件尾开始写*/

int main(void)
{
    int count;
    int fd;                                     /*文件描述符*/
    char write_buf[SIZE];                       /*写缓冲区*/
    const char *pathname=FILENAME;              /*指向需要打开文件的路径名*/
    if((fd=open(pathname,FLAGS))==-1)           /*调用 open 函数打开文件*/
      {
```

```
                printf("error,open file failed!\n");
                exit(1);     /*出错退出*/
            }
        printf("OK,open file successful!\n");
        printf("Begin write:\n");
        gets(write_buf);
        count = strlen(write_buf);                    /*要写入文件的数据的字节数*/
        if (write(fd, write_buf, count)==-1)
    {
            printf("error,write file failed!\n");
            exit(1);     /*写出错，退出*/
    }
        printf("OK,write %d strings to file!\n",count);
    return 0;
    }
```

在程序运行之前，先来看一下/home/zhangfan/hello 文件中的内容，使用 cat 命令：

```
# cat /home/zhangfan/hello
Hello! I like Linux C program!
```

使用 gcc 编译 write_file.c，并生成可执行文件 write_file：

```
#gcc –o write_file write_file.c
```

运行程序，得到输出结果：

```
#./write_file
OK,open file successful!
Begin write:
I am doing Linux C programs! ↙(↙表示回车)
OK,write 28 strings to file!
```

我们看到，用户从终端输入了“I am doing Linux C programs!”字符串。这时，再一次查看该文件的内容：

```
# cat /home/zhangfan/hello
Hello! I like Linux C program!
I am doing Linux C programs!
```

终端输入的字符成功追加在了 hello 文件的尾端。

而下面这个例子是 write 和 lseek 函数综合应用的例子，代码如程序 6.4 所示，hole.c 创建了一个具有空洞的文件。

说 明

当定位到超出文件尾端之后，对于新写入的数据需要分配磁盘块，但是对于原文件尾端和新开始写位置之间的部分则不需要分配磁盘块，这会产生空洞文件。文件中的空洞并不要求在磁盘上占有存储区，具体处理方式与文件系统的实现有关。

【程序 6.4】创建一个具有空洞的文件：hole.c。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define FILENAME "/home/zhangfan/test"        /*要进行操作的文件*/
#define FLAGS O_WRONLY | O_CREAT | O_TRUNC
/*定义参数 flags：以读写方式打开文件，向文件添加内容时从文件尾开始写*/
#define MODE 0600           /*定义参数 MODE：文件所有者读写方式*/

int main(void)
{
char buf1[ ]={"abcdefghij"};       /*缓冲区 1，长度为 10*/
      char buf2[ ]={"1234567890"};   /*缓冲区 2，长度为 10*/
int fd;                             /*文件描述符*/
int count;
      const char *pathname=FILENAME;      /*指向需要进行操作的文件的路径名*/
      if((fd=open(pathname,FLAGS,MODE))==-1)     /*调用 open 函数打开文件*/
        {
            printf("error,open file failed!\n");
            exit(1);     /*打开文件出错，退出*/
        }
count = strlen(buf1);                   /*缓冲区 1 的长度*/
      if(write(fd,buf1,count)!=count)           /*调用 write 函数将缓冲区 1 的数据写入文件*/
{
          printf("error,write file failed!\n");
          exit(1);     /*写出错，退出*/
       }
      if(lseek(fd,50,SEEK_SET)==-1)
/*调用 lseek 函数定位文件，偏移量为 50，从文件开头计算偏移值*/
       {
            printf("error,lseek failed!\n");
            exit(1);     /*定位出错，退出*/
              }
      count = strlen(buf2);                   /*缓冲区 2 的长度*/
      if(write(fd,buf2,count)!=count)           /*调用 write 函数将缓冲区 2 的数据写入文件*/
             {
            printf("error,write file failed!\n");
            exit(1);     /*写出错，退出*/
              }
      return 0;
}
```

使用 gcc 编译 hole.c，并生成可执行文件 hole：

```
#gcc –o hole hole.c
```

运行程序：

```
#./hole
```

程序首先打开(或创建)文件/home/zhangfan/test，接着调用 write 函数将缓冲区 1 的数据写入文件，此时写入的数据长度为 10，然后调用 lseek 定位文件，文件偏移量为 50，并从文件开头计算偏移值，最后调用 write 将缓冲区 2 的数据也写入文件，写入的数据长度也为 10。

在第二次写入数据时，中间 40 字节的内容为空(空洞)，文件的总长度为 60。成功运行程序后，利用 ls 和 od 命令可以检验出：

```
# ls -l /home/zhangfan/test
-rw-------    1 root    root         60   9 月  20 09:57 /home/zhangfan/test
# od -c /home/zhangfan/test
0000000   a   b   c   d   e   f   g   h   i   j  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0000060  \0  \0   1   2   3   4   5   6   7   8   9   0
0000074
```

使用 od 命令观察该文件的实际内容。命令行中的-c 标志表示以字符方式打印文件内容。从中可以看到，文件中间的 40 个未写字节都被读成 0。每一行开始的一个七位数是以八进制形式表示的字节位移量。

6.3 文件的属性操作

Linux 的文件系统具有比较复杂的属性，包括文件访问权限、文件所有者、文件名本身、文件长度等。本节介绍操作这些属性的函数调用。

6.3.1 改变文件访问权限

chmod、fchmod 这两个函数使用户可以更改现存文件的存取许可权：

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod (const char *pathname, mode_t mode);
int fchmod (int fd, mode_t mode);
```

两个函数的返回：若成功则为 0，若出错则为-1。

chmod 函数在指定的文件上进行操作，pathname 指定了这个文件的绝对路径名或相对路径名；而 fchmod 函数则对已打开的文件进行操作，fd 是这个打开文件的描述符。而参数 mode 的含义与表 6.5 相同。

提 示

为了改变一个文件的访问许可权位，进程的有效用户 ID 必须等于文件的所有者(User)，或者该进程必须具有超级用户许可权。

chmod 和 fchmod 的使用是比较简单的，例如我们在程序 6.4 中创建了一个具有空洞的文

本文件 test，它的访问权限为文件所有者可读可写，现在使用 chmod 来更改它的权限。代码如程序 6.5 所示。

【程序 6.5】 使用 chmod 函数改变文件访问权限：change_mode.c。

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>

#define FILENAME "/home/zhangfan/test"        /*要进行操作的文件*/
#define MODE 0755
 /*定义参数 MODE：文件所有者读、写、执行；组读、执行；其他读、执行方式*/

int main(void)
{
      const char *pathname=FILENAME;     /*指向需要进行操作的文件的路径名*/
      if(chmod (pathname,MODE)==-1)        /*调用 chmod 函数改变文件权限*/
        {
            printf("error,change failed!\n");
            exit(1);      /*出错退出*/
              }
      printf("OK,change successful!\n");
      return 0;
}
```

使用 gcc 编译 change_mode.c，并生成可执行文件 change_mode：

```
#gcc -o change_mode change_mode.c
```

运行程序，得到输出结果：

```
# ./change_mode
OK,change successful!
```

此时使用 ls -l 命令查看文件/home/zhangfan/test 的信息：

```
# ls -l /home/zhangfan/test
-rwxr-xr-x      1 root        root              60    9 月  20 09:57 /home/zhangfan/test
```

从中可以看到，显示的文件的执行权限信息与程序中的“MODE”参数值 0755(十六进制数)是吻合的。

提 示

在实际应用中，通常需要将某个文件的访问权限改为文件所有者读、写、执行；组读、执行；其他读、执行的方式，那么，在 Shell 中最常用的一个命令就是：

```
#chmod 755 filename
```

755 就代表了文件权限位对应的十六进制值。

6.3.2 改变文件所有者

用户可以通过系统调用 chown、fchown 和 lchown 来改变一个文件的所有者识别号和用户组识别号，函数说明如下：

```
#include <sys/types.h>
#include <unistd.h>
int chown(const char *pathname, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *pathname, uid_t owner, gid_t group);
```

3 个函数的返回：若成功则为 0，若出错则为-1。

- chown：修改指定文件的所有者，参数 pathnamee 指定了该文件的绝对路径名或相对路径名，owner 表示新赋予该文件的所有者标识号，group 表示新赋予该文件的组标识号。
- fchown：修改已打开文件的所有者，fd 为该文件描述符，owner 和 group 参数含义与 chown 相同。
- lchown：针对符号链接文件，参数与 chown 相同。需要注意的是，lchown 更改符号链接文件本身的所有者，而不是该符号链接所指向的文件。

6.3.3 重命名

一个普通文件或一个目录文件都可以被重命名，调用 rename 函数，它的说明如下：

```
#include <stdio.h>
int rename (const char *oldname, const char *newname);
```

返回：若成功则为 0，若出错则为-1。

rename 会将参数 oldname 所指定的文件名称改为参数 newname 所指定的文件名称。若 newname 所指定的文件已存在，则会被删除。当 oldname 和 newname 指向同一个文件时，rename 调用不做任何操作而成功返回。

rename 调用是否成功，将与 oldname 指向普通文件还是目录文件，newname 所表示的文件是否存在，若存在是普通文件还是目录文件等情况都有关系，表 6.6 列出了它们之间的关系。

表 6.6 rename 参数的不同情况

oldname 指向	newname 所示文件不存在	newname 所示文件存在	
		newname 指向普通文件	newname 指向目录文件
普通文件	文件成功被重命名	newname 被删除，原名为 oldname 的文件被重命名为 newname	错误
目录文件	文件成功被重命名	错误	若该目录文件为空目录则被删除，oldname 被重命名；否则出错

对于重命名目录文件的情况，有一点需要注意是，newname 不能包含有 oldname 的路径前缀，也就是说，不能将一个目录文件重命名为它的子文件。

6.3.4 修改文件长度

stat 结构体的成员 st_size(稍后介绍 stat)包含了以字节为单位的文件的长度。此字段只对普通文件、目录文件和符号链接文件有意义。

对于普通文件，其文件长度可以是 0，在读这种文件时，将得到文件结束指示。

对于目录，文件长度通常是一个数，例如 16 或 512 的整倍数，我们将在 6.5.1 小节中说明读目录操作。

对于符号链接，文件长度是在文件名中的实际字节数。例如：

```
lrwxrwxrwx   1   root          7   Sep 25 07:14   lib -> usr/lib
```

其中，文件长度 7 就是路径名 usr/lib 的长度(注意，因为符号链接文件长度总是由 st_size 指示，所以符号连接并不包含通常 C 语言用做名字结尾的 NULL 字符)。

有时我们需要在文件尾端处截去一些数据以缩短文件。将一个文件的长度截短为 0 是一个特例，用 O_TRUNC 标志(见表 6.4)可以做到这一点。为了截短文件可以调用函数 truncate 和 ftruncate。它们的说明如下：

```
#include <sys/types.h>
#include <unistd.h>
int truncate (const char *pathname, off_t len);
int ftruncate (int fd, off_t len);
```

两个函数返回：若成功则为 0，若出错则为-1。

这两个函数将由路径名 pathname 或打开文件描述符 fd 指定的一个现存文件的长度截短为 len。如果该文件以前的长度大于 len，则超过 len 以外的数据就不能再存取。如果以前的长度小于 len，则其后果与系统有关。如果某个实现的处理是扩展该文件，则在以前的文件尾端和新的文件尾端之间的数据将读成 0。

6.4 文件的其他操作

在了解了文件的创建、打开、读写、属性等操作之后，本节简要向读者介绍其他的一些常用文件操作，包括获取文件信息、改变文件性质等。

6.4.1 stat、fstat 和 lstat 函数

Linux 系统中的所有文件都有一个与之对应的索引节点，该节点中包含了文件的相关信息。这些信息被保存在 stat 结构体中，可以通过调用下面 3 个 stat 函数来返回文件的信息，说明如下：

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *pathname, struct stat *sbuf);
int fstat(int fd, struct stat *sbuf);
int lstat(const char *pathname,, struct stat *sbuf);
```

3 个函数的返回：若成功则为 0，若出错则为-1。

使用 stat 函数最多的就是 ls -l 命令，用其可以获得有关一个文件的所有信息。

同 chown 函数类似，stat 返回指定文件的信息结构，参数 pathname 指定了该文件的绝对路径名或相对路径名，fstat 函数获得已在描述符 fd 上打开的文件的有关信息。lstat 函数类似于 stat，但是当命名的文件是一个符号链接时，lstat 返回该符号链接的有关信息，而不是由该符号连接引用的文件的信息。

第二个参数是个指针，它指向一个我们应提供的结构。这些函数填写由 sbuf 指向的结构。该结构的基本形式如下：

```
struct stat
{
    mode_t  st_mode;          /*file type & mode (permission)*/
    ino_t   st_ino;           /*i-node number (serial number)*/
    dev_t   st_dev;           /*device number (filesystem)*/
dev_t   st_rdev;          /* device number for special files*/
    nlink_t  st_nlink;        /*number of links*/
    uid_t   st_uid;           /*user ID of owner*/
    gid_t   st_gid;           /*group ID of owner*/
    off_t   st_size;          /*size in bytes, for regular files*/
    time_t  st_atime;         /*time of last access*/
    time_t  st_mtime;         /* time of last modification*/
    time_t  st_ctime;         /* time of last file station change*/
    unsigned long st_blksize;  /*best I/O block size*/
    unsigned long st_blocks;  /*number of 512-byte block allocated*/
};
```

在上面的代码中可以看到，除最后两个成员以外，其他各成员都为基本系统数据类型。读者在本章前面的内容也看到过这样的数据类型，例如 mode_t, off_t。下面向读者介绍 UNIX/Linux 系统下的基本系统数据类型。

在历史上，某些 UNIX 变量已与某些 C 数据类型联系在一起，例如，历史上主、次设备号存放在一个 16 位的短整型中，8 位表示主设备号，另外 8 位表示次设备号。但是，很多较大的系统需要用多于 256 个值来表示其设备号，于是，就需要有一种不同的技术。(SVR4 用 32 位表示设备号：14 位用于主设备号，18 位用于次设备号。)

头文件< sys/types.h >中定义了某些与实现有关的数据类型，它们被称之为基本系统数据类型(primitive system data type)。有很多这种数据类型定义在其他头文件中。在头文件中这些数据类型都是用 C 的 typedef 设施来定义的。它们绝大多数都以_t 结尾。表 6.7 中列出了常见的基本系统数据类型。用这种方式定义了这些数据类型后，在 Linux 下开发和编译 C 程序时，就不再需要考虑随系统不同而改变的实施细节了。

表 6.7 UNIX/Linux 常见基本系统数据类型

类　型	说　明
caddr_t	内存地址
clock_t	时钟滴答计数器(进程时间)

(续表)

类　　型	说　　明
comp_t	压缩的时钟滴答
dev_t	设备号(主和次)
fd_set	文件描述符集
fpos_t	文件位置
gid_t	数值组 ID
ino_t	i 节点编号
mode_t	文件类型，文件创建方式
n1ink_t	目录项的连接计数
off_t	文件长度和位移量(带符号的)
pid_t	进程 ID 和进程组 ID(带符号的)
ptrdiff_t	两个指针相减的结果(带符号的)
r1im_t	资源限制
sig_atomic_t	能原子地存取的数据类型
sigset_t	信号集
size_t	对象(例如字符串)长度(不带符号的)
ssize_t	返回字节计数的函数(带符号的)(如 read，write)
time_t	日历时间的秒计数器
uid_t	数值用户 ID
wchar_t	能表示所有不同的字符码

6.4.2　dup 和 dup2 函数

下面两个函数都可用来复制一个现存的文件描述符：

```
#include <unistd.h>
int dup (int fd);
int dup2 (int fd, int fd2);
```

两个函数的返回：若成功为新的文件描述符，若出错为-1。

由 dup 返回的新文件描述符一定是当前可用文件描述符中的最小数值。用 dup2 则可以用 fd2 参数指定新描述符的数值。如果 fd2 已经打开，则先将其关闭。若 fd 等于 fd2，则 dup2 返回 fd2，而不关闭它。通常使用这两个系统调用来重定向一个已打开的文件描述符。

6.4.3　fcntl 函数

fcntl 函数可以改变已经打开文件的性质，它的说明如下：

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
```

```
int fcntl (int fd, int cmd);
int fcntl (int fd, int cmd, long arg);
```

返回：若成功则依赖于 cmd，若出错为-1。

参数 cmd 表示此调用所要执行的操作，其相应的取值和执行的操作如表 6.8 所示。参数 arg 是可选的，对应于 cmd 的某些可取值，用于执行特殊的操作。

表 6.8　cmd 取值及相应操作

cmd 取值	相 应 操 作
F_DUPFD	复制一个现存的文件描述符
F_GETFD	获得文件描述符标记
F_SETFD	设置文件描述符标记
F_GETFL	获得文件状态标志
F_SETFL	设置文件状态标志
F_GETOWN	获得异步 I/O 有权
F_SEOWN	设置异步 I/O 有权
F_GETLK	获得记录锁
F_SETLK	设置记录锁，不等待
F_SETLKW	设置记录锁，必要时等待

鉴于 fcntl 函数在实际程序开发过程中的使用并不多见，并且 fcntl 调用能完成的大部分操作都可以用其他的调用来完成，这里不再详细阐述。

6.4.4　sync 和 fsync 函数

传统的 UNIX 实现在内核中设有缓冲存储器，大多数磁盘 I/O 都通过缓存进行。当将数据写到文件上时，通常该数据先由内核复制到缓存中，如果该缓存尚未写满，则并不将其排入输出队列，而是等待其写满或者当内核需要重用该缓存以便存放其他磁盘块数据时，再将该缓存排入输出队列，然后待其到达队首时，才进行实际的 I/O 操作。这种输出方式被称之为延迟写(delayed write)。延迟写减少了磁盘读写次数，但是却降低了文件内容的更新速度，使得欲写到文件中的数据在一段时间内并没有写到磁盘上。当系统发生故障时，这种延迟可能造成文件更新内容的丢失。为了保证磁盘上实际文件系统与缓存中内容的一致性，UNIX 系统提供了 sync 和 fsync 两个系统调用函数。它们的说明如下：

```
#include <unistd.h>
void sync(void);
int fsync(int fd);
```

返回：若成功则为 0，若出错则为-1。

sync 只是将所有修改过的块的缓存排入写队列，然后返回，它并不等待实际 I/O 操作结束。系统进程(通常称为 update)一般每隔 30 秒调用一次 sync 函数。这就保证了定期刷新内核的块缓存。命令 sync(1)也调用 sync 函数。

函数 fsync 只引用单个文件(由文件描述符 fd 指定)，它等待 I/O 结束，然后返回。fsync 可

用于数据库这样的应用程序，它确保修改过的块立即写到磁盘上。

比较一下 fsync 和 O_SYNC 标志(参见表 6.4)。当调用 fsync 时，它更新文件的内容，而对于 O_SYNC，则每次对文件调用 write 函数时就更新文件的内容。

6.5 特殊文件的操作

Linux 的文件类型除了普通文件以外，还包括目录文件、链接文件、管道文件、设备文件、套接字文件等。本节介绍目录文件、链接文件、管道文件的操作，以及设备文件的简单介绍，套接字文件将在网络编程一章中讲解。

6.5.1 目录文件的操作

目录就是一个文件夹，文件可以存放在目录中，在 Linux 系统中，目录也作为文件来处理，本小节向读者介绍操作目录文件的相关函数调用。

1. mkdir 和 rmdir 函数

(1) 创建目录

mkdir 函数用于创建目录，函数说明如下：

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir (const char *pathname, mode_t mode);
```

返回：若成功则为 0，若出错则为-1。

此函数创建一个新的空目录。参数 pathname 和 mode 的含义不再赘述。需要强调的一点是，对于目录文件通常需要至少设置 1 个执行许可权位，以允许存取该目录中的文件名。

(2) 删除空目录

rmdir 函数用于删除空目录：

```
#include <unistd.h>
int rmdir (const char *pathname);
```

返回：若成功则为 0，若出错则为-1。

如果此调用使目录的连接计数成为 0，并且也没有其他进程打开此目录，则释放由此目录占用的空间。如果在连接计数达到 0 时，有一个或几个进程打开了此目录，则在此函数返回前删除最后一个连接。另外，在此目录中不能再创建新文件。但是在最后一个进程关闭它之前并不释放此目录(即使某些进程打开该目录，它们在此目录下，也不能执行其他操作，因为为使 rmdir 函数成功执行，该目录必须是空的)。

2. opendir、closedir 和 readdir 函数

(1) 打开目录

打开目录的系统调用为 opendir，函数原型如下：

```
#include <sys/types.h>
#include <dirent.h>
```

```
DIR *opendir (const char *pathname);
```

返回：若成功则为指针，若出错则为 NULL。

opendir 的返回值为 DIR 类型，是用于指向目录文件的结构指针。

DIR 结构是一个内部结合 3 个函数来保存正被读的目录的有关信息，其作用类似于 FILE 结构。FILE 结构由标准 I/O 库维护(我们将在第 7 章中对它进行说明)。

(2) 关闭目录

关闭目录的系统调用为 closedir，函数原型如下：

```
#include <sys/types.h>
#include <dirent.h>
int closedir (DIR *dp);
```

返回：若成功则返回 0，若出错则为-1。

参数 dp 是 DIR 类型的指针，指向要关闭的目录文件，该指针由 opendir 调用时返回。

(3) 目录文件的读取

对某个目录具有存取许可权的任一用户都可读该目录，但是只有内核才能写目录(防止文件系统发生混乱)。参考 6.1.3 节，一个目录的写许可权位和执行许可权位决定了在该目录中能否创建新文件及删除文件，它们并不表示能否写目录本身。目录文件的读取调用说明如下：

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir (DIR *dp);
```

返回：若成功则为指针，若在目录尾或出错则为 NULL。

参数 dp 指向要读取的目录，函数返回值为指向 dirent 结构体的指针。dirent 定义在头文件 <dirent.h>中：

```
struct dirent
{
ino_t d_ino;                        /*i-node number*/
char d_name[NAME_MAX + 1];          /*null-terminated filename*/
}
```

其中 d_ino 用于表示该目录的节点号，d_name 用于存放此目录链接的文件名。当目录中没有更多链接时，其值为 0。

3. chdir、fchdir 和 getcwd 函数

每个进程都有一个当前工作目录，此目录是搜索所有相对路径名的起点(不以斜线开始的路径名为相对路径名)。当用户登录到 Linux 系统时，其当前工作目录通常是口令文件(/etc/passwd)中该用户登录项的第 6 个字段——用户的起始目录。当前工作目录是进程的一个属性，起始目录则是登录名的一个属性。进程调用 chdir 或 fchdir 函数可以更改当前工作目录。函数说明如下：

```
#include <unistd.h>
int chdir (const char *patnname);
int fchdir (int fd);
```

两个函数的返回：若成功则为 0，若出错则为-1。

在这两个函数中，可以分别用 patnname 或打开文件描述符来指定新的当前工作目录。我们需要一个函数，它从当前工作目录(.目录项)开始，用“.目录项”找到其上一级的目录，然后读其目录项，直到该目录项中的 i 节点编号数与工作目录 i 节点编号数相同，这样就找到了其对应的文件名。按照这种方法，逐层上移，直到遇到根，这样就得到了当前工作目录的绝对路径名。函数 getcwd 就是提供这种功能的，它的说明如下：

```
#include <unistd.h>
char *getcwd (char *buf, size_t size);
```

返回：若成功则返回 buf，若出错则返回 NULL。

向此函数传递两个参数，一个是缓存地址 buf，另一个是缓存的长度 size。该缓存必须有足够的长度以容纳绝对路径名再加上一个 NULL 终止字符，否则返回出错。

程序 6.6 将当前工作目录更改至一个特定的目录(由用户输入)，然后调用 getcwd，最后打印当前工作目录。代码如 change_path.c。

【程序 6.6】改变并获取当前工作目录：change_path.c。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

#define SIZE 30                    /*定义缓冲区大小*/

int main(void)
{
char newpath[SIZE];
char buf[SIZE];
            printf("Input the new pathname[<30 strings]:");
     gets(newpath);
     if(chdir(newpath) == -1)        /*调用 chdir 函数改变当前工作目录*/
        {
             printf("error,change directory failed!\n");
             exit(1);     /*出错退出*/
               }
     printf("OK,change directory successful!\n");
          if(getcwd(buf,SIZE)==NULL)   /*调用 getcwd 函数获取当前工作目录*/
            {
                  printf("error,getcwd failed!\n");
             exit(1);     /*出错退出*/
             }
printf("cwd = %s\n",buf);
          return 0;
}
```

运行程序之前，先来看看当前的工作目录，使用 pwd 命令可以查看：

```
# pwd
/home/zhangfan/CODE
```

使用 gcc 编译 change_path.c，并生成可执行文件 change_path：

```
#gcc –o change_path change_path.c
```

运行程序，得到输出结果：

```
# ./change_path
Input the new pathname[<30 strings]:/root ↙(↙表示回车)
OK,change directory successful!
cwd = /root
```

使用 pwd 命令再次查看当前工作目录：

```
# pwd
/root
```

由此可见，程序已成功将系统的当前工作目录切换至/root 目录(用户任意输入的小于 30 个字符的有效目录)，这是由 chdir 调用实现的。而 getcwd 成功获取了当前的工作目录，即输出了“cwd = /root”。

当一个应用程序需要在文件系统中返回到其工作的起点时，getcwd 函数是很有用的。在更换工作目录之前，我们可以调用 getcwd 函数先将其保存起来。在完成了处理之后，就可将从 getcwd 获取的路径名作为调用参数传递给 chdir，这样就返回到了文件系统中的起点。

fchdir 函数向我们提供了一种完成此任务的便捷方法。在更换到文件系统不同的位置前，无须调用 getcwd 函数，而是使用 open 打开当前工作目录，然后保存文件描述符。当希望回到原工作目录时，只要简单地将文件描述符传递给 fchdir。

6.5.2 链接文件的操作

链接文件是 Linux 系统中的一种特殊文件，它实际上是指向一个现实存在的文件的链接。链接文件又分为硬链接文件和符号链接文件，下面分别给出这两种情况的相关的系统调用。

1. 硬链接

(1) 创建链接

创建一个向现存文件链接的方法是使用 link 函数，函数原型如下：

```
#include <unistd.h>
int link (const char *pathname1, const char *pathname2);
```

返回：若成功则为 0，若出错则为-1。

此函数创建一个新目录项 pathname2，它引用现存文件 pathname1。若 pathname2 已经存在，则返回出错。

硬链接要求 pathname1 和 pathname2 所指向的路径名应当在同一个文件系统中。另外，只有超级用户 root 才可以创建指向一个目录的新链接。

(2) 删除链接

为了删除一个现存的目录项，可以调用 unlink 函数。函数原型如下：

```
#include <unistd.h>
int unlink (const char *pathname);
```

返回：若成功则为 0，若出错则为-1。

此函数删除目录项，并将由 pathname 所引用的文件的连接计数减 1。如果该文件还有其他连接，则仍可通过其他连接存取该文件的数据。如果出错，则不对该文件进行任何更改。

我们在前面已经提及，为了解除对文件的连接，必须对包含该目录项的目录具有写和执行许可权，并且具备下面 3 个条件之一：

- 拥有该文件。
- 拥有该目录。
- 具有超级用户优先权。

只有当连接计数达到 0 时，该文件的内容才可被删除。另一个条件也阻止删除文件的内容——只要有进程打开了该文件，其内容也不能删除。关闭一个文件时，内核首先检查使该文件打开的进程计数。如果该计数达到 0，然后内核检查其连接计数，如果这也是 0，那么就删除该文件的内容。

unlink 的这种特性经常被程序用来确保即使是在程序崩溃时，它所创建的临时文件也会保留下来。进程用 open 或 creat 创建一个文件，然后立即调用 unlink。因为该文件仍旧是打开的，所以不会将其内容删除。只有当进程关闭该文件或终止时(在这种情况下，内核关闭该进程所打开的全部文件)，该文件的内容才被删除。

如果 pathname 是符号连接，那么 unlink 涉及的是符号连接而不是由该连接所引用的文件。超级用户可以调用带参数 pathname 的 unlink 指定一个目录，但是通常不使用这种方式，而使用函数 rmdir。

我们也可以用 remove 函数解除对一个文件或目录的连接。对于文件，remove 的功能与 unlink 相同。对于目录，remove 的功能与 rmdir 相同。

```
#include <stdio.h>
int remove(const char *pathname);
```

返回：若成功则为 0，若出错则为-1。

2. 符号链接

符号链接是对一个文件的间接指针，它与前面所述的硬链接有所不同，硬链接直接指向文件的 i 节点。引进符号链接的原因是为了避免硬链接的一些限制：一是硬链接通常要求链接和文件位于同一文件系统中；二是只有超级用户才能创建到目录的硬链接。对符号链接及它指向什么没有文件系统限制，任何用户都可创建指向目录的符号链接。符号链接一般用于将一个文件或整个目录结构移到系统中其他某个位置。

当使用以名字引用一个文件的函数时，应当了解该函数是否处理符号链接功能。也就是是否跟随符号链接到达它所链接的文件。若该函数处理符号链接功能，则该函数的路径名参数引用由符号链接指向的文件。否则，一个路径名参数引用链接本身，而不是由该链接指向的文件。

使用符号链接可能在文件系统中引入循环。大多数查找路径名的函数在这种情况发生时都返回值为 ELOOP 的 errno。考虑下列命令序列：

```
#mkdir foo                  创建一个新目录
#touch foo/test             创建一个长度为 0 文件
#ln -s ../foo foo/testdir   创建符号链接
#ls -1 foo
-rw-rw-r--    1 root   root     0    9 月  20 19:53 test
lrwxrwxrwx    1 root   root     6    9 月  20 19:53 testdir -> ../foo
```

这创建了一个目录 foo，它包含了一个名为 aaa 的文件及一个指向 foo 的符号链接。在图 6.3 中显示了这种结果，图中以圆表示目录，以正方形表示一个文件。如果我们写一段简单的程序，使用标准库函数 ftw(3)以降序遍历文件结构，打印每个遇到的路径名，则其输出是：

```
foo
foo/test
foo/testdir
foo/testdir/test
foo/testdir/testdir
foo/testdir/testdir/test
foo/testdir/testdir/testdir
foo/testdir/testdir/testdir/test
(更多行，直至 ftw 出错返回，此时 errno 值为 ELOOP)
ftw returned -1:Too many levels of symbolic links
```

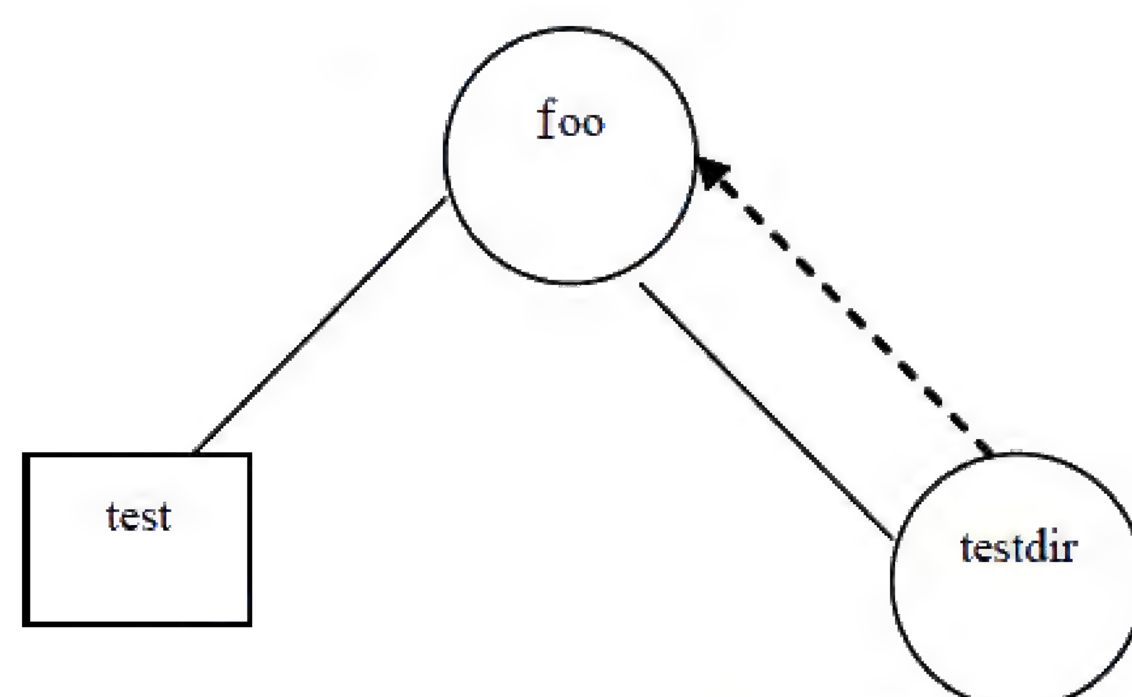


图 6.3　构成循环的符号链接 testdir

symlink 函数用于创建一个符号链接，函数原型如下：

```
#include <unistd.h>
int symlink (const char *actualpath, const char *sympath);
```

返回：若成功则为 0，若出错则为-1。

该函数创建了一个指向 actualpath 的新目录项 sympath，在创建此符号链接时，并不要求 actualpath 已经存在。并且，actualpath 和 sympath 并不需要位于同一文件系统中。

因为 open 函数往往跟随符号链接，所以需要有一种方法来打开该链接本身，并读取该链接中的名字。readlink 函数提供了这种功能，它用于打开一个链接并获取该链接的名字，函数原型如下：

```
#include <unistd.h>
int readlink (const char *pathname, char *buf, int bufsize);
```

返回：若成功则返回读取的字节数，若出错则为-1。

参数 pathname 为所要查看的链接，参数 buf 为一个字符串指针，获取的相关信息将存储在 buf 所指向的缓冲区中，参数 bufsize 表示该缓冲区的大小。

此函数组合了 open、read 和 close 的所有操作，如果此函数成功，则它返回读入 buf 的字节数。注意，在 buf 中返回的符号链接的内容不以 NULL 字符终止。

6.5.3 管道文件的操作

管道文件也是Linux系统中的一种很特殊的文件，主要用于不同进程间的数据和信息传递。一个进程将要传递的数据或信息写入管道的一端，另一进程则从管道的另一端取得所需的数据或信息。

管道的创建可以使用 pipe 函数来实现：

```
#include <stdio.h>
int pipe ( int filedes[2] );
```

返回：若成功则返回 0，出错则为-1。

参数 filedes 是一个含有两个元素的数组。当调用 pipe 成功创建管道后，将返回两个文件描述符，分别到管道的两端。

> **说 明**
>
> 通常情况，pipe 与 fork、dup2 及 execve 等函数配合使用，来为被重定向 I/O 的其他程序创建管道。

关于管道文件的具体使用在第 10 章“进程间通信中”会详细介绍。

6.5.4 设备文件

在 6.1.2 小节中已经向读者提到了设备文件。设备文件是 Linux 系统中一类非常特殊的文件，正是由于它的存在，使得用户可以十分方便地访问外部设备。Linux 系统为外部设备提供了一种统一标准接口，将外部设备视为一种特殊的文件，可以像访问文件一样访问一个外部设备。这就使得 Linux 系统可以很方便地适应不断发展的外部设备。

在实际操作中，几乎总是不可避免地要用到设备文件。由于在 Linux 系统中，所有的外部设备都被看作是目录/dev 下的一个文件，所以前面讲的各种关于文件的系统调用都可以在外部设备上使用，因而可以很方便地使用基于文件描述符的 I/O 操作实现外部设备的读写。

但也要注意一些特殊的设备，它们在某些方面具有其自身的特殊性。例如，对于像磁带这样的外存，只可以顺序地对它进行读写访问。而用于实现随机读写的系统调用 lssek 对磁带来说是无效的。同时当系统调用 close 时，将导致磁带的回绕。

6.6 本章小结

本章介绍了 Linux 的文件系统和基于文件描述符的基本输入输出操作的相关函数调用，以

及与文件操作相关的函数，包括获取文件信息、修改文件属性、重命名文件等。同时还介绍了Linux 下特殊文件(目录文件、链接文件、管道文件和设备文件)的操作。熟悉 Linux 的文件系统及其操作的相关系统调用对于在 Linux 环境下编写 C 程序将有很大帮助。

实战演练

1. 执行“ls -l”命令，并将该命令的结果重定向到 filelist 文件中。
2. 执行“pwd”命令，并将该命令的结果追加重定向到 filelist 文件中。
3. 在 Shell 中输入一个不正确的命令，将产生一个标准出错信息，比如：

```
bash: xxx: command nor found
```

“xxx”表示我们输入的错误命令，试将该出错信息再次追加重定向到 filelist 文件中。

4. 编写一个程序，使用 open 函数以只读方式打开/usr/src/linux-2.4.20-8/init 目录下的 main.c 文件。

5. 编写一个程序，实现以下功能：先打开/home/zhangfan/test 文件(如果没有该文件，则创建它)，接着读取文件中的内容，然后向该文件写入数据，写入的数据由用户从键盘输入。

6. 编写一个程序，将第 3 题中生成的 filelist 文件的读写位置定位在第 10 个字节，并返回函数执行的结果。

7. 编写一个程序，在/home/zhangfan 目录下创建一个空目录/temp，该目录文件的访问权限为用户可读可写可执行，同组用户可读可执行，其他组用户可读可执行，并返回函数执行的结果。

8. 编写一个程序，改变第 3 题中生成的 filelist 文件的访问权限，要求文件所有者具有读、写、执行权限；同组用户具有读、执行权限；其他组用户具有读、执行权限，并返回函数执行的结果。

9. 编写一个程序，使用 rename 函数为/home/zhangfan 目录下的 hello.txt 文件重命名，将其重命名为 temp.txt。

10. 编写一个程序，使用 link 函数为/home/zhangfan/hello.txt 文件创建一个硬链接，新的目录项为/root/hello.txt。

7

基于流的I/O操作

本章介绍另一种执行输入输出操作的方法——基于流的 I/O 操作。流 I/O 是由 C 语言的标准函数库提供的，这些 I/O 可以代替系统中提供的 read 和 write 函数。事实上流 I/O 的内部封装了这两个基本的文件读写系统调用。使用流 I/O 在某些程度上来讲比基于文件描述符的 I/O 要更加简单、方便，因而在 C 程序开发中被广泛地使用。



本章内容：

◎ 流与缓存的概念。
◎ 流的打开与关闭操作。
◎ 基于各种不同方式的流的读写操作。

7.1 流与缓存

基于流的 I/O 操作与基于文件描述符的 I/O 操作过程十分相似，同样是使用相关的函数调用打开文件或设备，然后对文件进行读写，最后关闭文件。所以在进行流 I/O 操作的讲解之前，读者有必要先了解它的一些基础概念，体会两者的异同。

7.1.1 流和 FILE 对象

在上一章中，所有 I/O 函数都是针对文件描述符的。当打开一个文件时，即返回一个文件描述符，然后通过该文件描述符来进行后续的 I/O 操作。而对于标准 I/O 库，它们的操作则是围绕流(stream)进行的。当用标准 I/O 库打开或创建一个文件时，已使一个流与一个文件相结合。

当打开一个流时，标准 I/O 函数 fopen 返回一个指向 FILE 对象的指针。该对象通常是一个结构体，它包含了 I/O 库为管理该流所需要的所有信息，包括用于实际 I/O 的文件描述符，指向流缓存的指针，缓存的长度，当前在缓存中的字符数，出错标志等。

应用程序没有必要检验 FILE 对象。为了引用一个流，需将 FILE 指针作为参数传递给每个标准 I/O 函数。在实际应用中，并不需要对 FILE 结构的内容有所了解，只需要在调用 I/O 库函数对流进行操作时会使用它。在本书中，我们称指向 FILE 对象的指针(类型为 FILE *)为文件指针。

基于流的 I/O 操作过程可以大致归纳如下：对流进行操作的第一步是通过调用 fopen()函数将其打开，并返回一个 FILE 结构指针。当流成功打开以后，就可以调用相应的库函数对其进行 I/O 操作了。当完成操作后，需要执行清空缓冲区、保存数据等操作，然后将流关闭，这些工作可以通过调用 fclose()函数来完成。注意，如果不关闭流，可能造成数据的丢失。

7.1.2 标准输入、标准输出和标准出错

当使用流 I/O 时，有 3 个流会自动地打开：标准输入、标准输出和标准出错。在上一章中我们曾用文件描述符 STDIN_FILENO、STDOUT_FILENO 和 STDERR_FILENO 分别表示它们，这 3 个符号常量是定义在头文件<unistd.h>中的。

而在基于流的 I/O 操作中，我们是通过预定义文件指针 stdin、stdout 和 stderr 来引用标准输入、标准输出和标准出错的。这 3 个文件指针定义在头文件<stdio.h>中。

另外，和自动打开文件一样，标准输入、标准输出和标准错误输出也是自动关闭的。

7.1.3 缓存

基于流的操作最终会调用 read 或者 write 函数进行 I/O 操作。为了提高程序的运行效率，尽可能减少使用 read 和 write 调用的数量，流对象通常会提供缓冲区(也叫作缓存，两者概念完全等同)，以减少调用系统 I/O 库函数的次数。标准 I/O 提供了 3 种类型的缓存：

- 全缓存：直到缓冲区被填满，才调用系统 I/O 函数。对于读操作来说，直到读入的内容的字节数等于缓冲区大小或者文件已经到达结尾，才进行实际的 I/O 操作，将外存文件

内容读入缓冲区；对于写操作来说，直到缓冲区被填满，才进行实际的I/O操作，缓冲区内容写到外存文件中。磁盘文件通常是全缓冲的。

- 行缓存：直到遇到换行符“\n”，才调用系统 I/O 库函数。对于读操作来说，遇到换行符“\n”才进行 I/O 操作，将所读内容读入缓冲区；对于写操作来说，遇到换行符“\n”才进行 I/O 操作，将缓冲区内容写到外存中。由于缓冲区的大小是有限的，所以当缓冲区被填满时，即使没有遇到换行符“\n”，也同样会进行实际的 I/O 操作。标准输入 stdin 和标准输出 stdout 默认都是行缓冲的。
- 无缓存：没有缓冲区，数据会立即读入或者输出到外存文件和设备上。标准出错 stderr 是无缓冲的，这样保证错误提示和输出能够及时反馈给用户，供用户排除错误。

以上 3 种缓冲区分别定义为 3 个宏，它们同样位于头文件<stdio.h>，其定义如表 7.1 所示。

表 7.1　缓冲区类型的宏定义

缓冲区类型	定 义 的 宏
全缓存	_IO_FULL_BUF
行缓存	_IO_LINE_BUF
无缓存	_IO_UNBUFFERED

在使用表 7.1 所列的缓冲区类型宏时，应将文件流对象中的缓冲区标志与该宏进行逻辑“与”操作，通过判断结果是否为 0 即可知道该文件流的缓冲区属于何种类型了。程序 7.1 演示了如何得到文件流的缓冲区类型，该程序输出标准输出、标准输入和标准出错 3 个流的缓冲区类型、缓冲区大小等信息。代码如 buf_test.c 所示。

【程序 7.1】检测缓冲区类型和大小：buf_test.c。

```
#include <stdio.h>
int main(void)
{
    printf("stdin is ");
    if(stdin->_flags & _IO_UNBUFFERED) /*判断标准输入流对象的缓冲区类型*/
        printf("unbuffered\n");               /*无缓存*/
    else if(stdin->_flags & _IO_LINE_BUF)
        printf("line-buffered\n");             /*行缓存*/
    else
        printf("fully-buffered\n");            /*全缓存*/
    printf("buffer size is %d\n", stdin->_IO_buf_end - stdin->_IO_buf_base);
                                               /*打印缓冲区的大小*/
    printf("file discriptor is %d\n\n", fileno(stdin));  /*标准输入流的文件描述符*/
    printf("stdout is ");
    if(stdout->_flags & _IO_UNBUFFERED)    /*判断标准输出流对象的缓冲区类型*/
        printf("unbuffered\n");                /*无缓存*/
    else if(stdout->_flags & _IO_LINE_BUF)
        printf("line-buffered\n");             /*行缓存*/
    else
        printf("fully-buffered\n");            /*全缓存*/
    printf("buffer size is %d\n",stdout->_IO_buf_end - stdout->_IO_buf_base);
```

```
                                                    /*打印缓冲区的大小 */
    printf("file descriptor is %d\n\n",fileno(stdout));    /*标准输出流的文件描述符 */
    printf("stderr is ");
    if(stderr->_flags & _IO_UNBUFFERED)     /*判断标准出错流对象的缓冲区类型*/
        printf("unbuffered\n");              /*无缓存*/
    else if(stderr->_flags & _IO_LINE_BUF)
        printf("line-buffered\n");           /*行缓存*/
    else
        printf("fully-buffered\n");          /*全缓存*/
    printf("buffer size is %d\n", stderr->_IO_buf_end - stderr->_IO_buf_base);
/*打印缓冲区的大小*/
    printf("file descriptor is %d\n\n", fileno(stderr));    /*标准出错的文件描述符 */
    return 0;
}
```

使用 gcc 编译 buf_test.c，并生成可执行文件 buf_test：

```
#gcc –o buf_test buf_test.c
```

运行程序，得到输出结果：

```
# ./buf_test
stdin is fully-buffered
buffer size is 0
discriptor is 0

stdout is line-buffered
buffer size is 1024
discriptor is 1

stderr is unbuffered
buffer size is 0
discriptor is 2
```

将输入和输出重定向后，执行程序(当然，在当前工作目录下需要存在文件 in.txt、out.txt 和 err.txt)：

```
# ./buf_test < in.txt > out.txt 2 > err.txt
```

所有的输出信息被重定向到 err.txt 文本文件中，查看该文件的内容得到如下信息：

```
# cat err.txt
stdin is fully-buffered
buffer size is 0
discriptor is 0

stdout is fully-buffered
buffer size is 4096
discriptor is 1

stderr is unbuffered
buffer size is 0
discriptor is 2
```

可以看到，标准输出的缓冲区类型发生了变化，当输出被重定向到文件 err.txt 中时，标准输出由行缓冲变成了全缓冲，缓冲区的大小也由 1024 变成了 4096。稍后我们还将看到用程序的方法对缓冲区的属性(类型、大小等)进行设置。

另外，不论什么时候，标准出错都是无缓冲的。这样可以保证出错信息及时地输出给用户，供用户排除错误、解决问题。

7.1.4 对缓存的操作

在进行基于流的 I/O 操作时，缓存的使用将是不可或缺的。本小节将介绍使用缓存时遇到的常用操作。

1. 设置缓存的属性

缓存具有自己的属性，其属性值包括缓冲区的类型和缓冲区的大小。当调用 fopen 函数打开一个流时，就开辟了所需的缓冲区，系统通常会赋予其一个默认的属性值。这些默认值是系统默认的、使用频率最高的值。实际使用时，也可以根据自己的需要来设定缓冲区的属性值，可以通过调用如下的函数：

```
#include <stdio.h>
void setbuf(FILE *fp, char *buf);
void setbuffer(FILE *fp, char *buf, size_t size);
void setlinebuf(FILE *fp);
int setvbuf(FILE *fp, char *buf, int mode, size_t size);
```

前 3 个函数没有返回值。setvbuf 函数的返回：若成功则为 0，若出错则为非 0。

这 4 个函数都用于对缓存的属性进行设置，它们都涉及同一个参数 fp，这是一个 FILE 结构指针，应指向一个已经打开的流。也就是说，在调用上述库函数时，流必须是打开的。

同时，各个函数具有不同的功能和特点：

- setbuf 用于将缓冲区设置为全缓存或无缓存。参数 buf 为指向缓冲区的指针，当 buf 指向一个真实的缓冲区地址时，此函数将缓冲区设置为全缓存，其大小由预定义常数 BUFSIZ 指定；当 buf 为 NULL 时，则设定为无缓存。所以此函数一般可当作激活或禁止缓冲区的开关。
- setbuffer 的功能与使用方法与 setbuf 函数相似，其区别是由程序员自行指定缓冲区的大小，由参数 size 指定。
- setlinebuf 专用于将缓冲区设定为行缓存。
- setvbuf 函数比较灵活，它可以很方便地设置缓存的属性，是以上 4 个函数中最基本的，前三个函数的功能都可以用此函数来实现。参数 fp、buf 和 size 的含义与 setbuffer 函数相同，参数 mode 用于指定缓冲区的类型，其值可取为_IOFBF(全缓存)、_IOLBF(行缓存)、_IONBF(无缓存)。这 3 个常量也是定义在<stdio.h>中。其中，当 mode 设定为_IONBF 时，buf 和 size 的值都是无效的。

另外，前 3 个函数是为了兼容 BSD 的老版本代码而定义的，没有返回值。建议在写新代码时使用 setvbuf 函数来代替这 3 个函数。

提 示

最好在将流打开但还未对流执行其他操作时设定流的属性。因为对流的各种操作都是和缓冲区的属性紧密相关的，改变缓冲区的属性会对所执行的操作产生意想不到的影响。

下面的实例说明了如何更改某一个流的缓冲区的属性。程序 7.2 先将标准输入的缓冲区类型设为无缓冲，并打印其信息，然后再将缓冲区设为全缓存，并相应地设置了缓冲区的大小，最后打印此时的缓存信息。代码如 buf_set.c 所示。

【程序 7.2】设置缓冲区属性：buf_set.c。

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 1024                          /*缓冲区的大小*/
int main(void)
{
    char buf[SIZE];                        /*缓冲区*/
    if(setvbuf(stdin, buf, _IONBF, SIZE)!=0)    /*将标准输入的缓冲类型设为无缓冲*/
      {
        printf("error!\n");
        exit(1);                           /*出错退出*/
      }
    printf("OK, set successful!\n");
    printf("stdin is ");                   /*打印缓冲区信息*/
    if(stdin->_flags & _IO_UNBUFFERED) /*判断标准输入流对象的缓冲区类型*/
        printf("unbuffered\n");
    else if(stdin->_flags & _IO_LINE_BUF)
        printf("line-buffered\n");
    else
        printf("fully-buffered\n");
    printf("buffer size is %d\n", stdin->_IO_buf_end - stdin->_IO_buf_base);
/*打印缓冲区的大小*/
    printf("file discriptor is %d\n", fileno(stdin));   /*输出文件描述符*/
    if(setvbuf(stdin, buf, _IOFBF, SIZE)!=0)
      { /*将标准输入的缓冲类型设为全缓冲，缓存大小为 1024*/
        printf("error!\n");
        exit(1);        /*出错退出*/
      }
    printf("OK, change successful!\n");
    printf("stdin is ");                   /*打印缓冲区信息*/
    if(stdin->_flags & _IO_UNBUFFERED)   /*判断标准输入流对象的缓冲区类型*/
        printf("unbuffered\n");
    else if(stdin->_flags & _IO_LINE_BUF)
        printf("line-buffered\n");
    else
        printf("fully-buffered\n");
    printf("buffer size is %d\n", stdin->_IO_buf_end - stdin->_IO_buf_base);
/*打印缓冲区的大小*/
    printf("file discriptor is %d\n", fileno(stdin));   /*输出文件描述符 */
    return 0;
}
```

使用 gcc 编译 buf_set.c，并生成可执行文件 buf_set：

```
#gcc –o buf_set buf_set.c
```

运行程序，得到输出结果：

```
# ./ buf_set
OK, set successful!
stdin is unbuffered
buffer size is 1
file discriptor is 0
OK, change successful!
stdin is fully-buffered
buffer size is 1024
file discriptor is 0
```

从中可以看到，stdin 的缓冲区类型发生了变化。在第一次将其设为 unbuffered 时，参数 buf 和常量 SIZE 并没有起作用。当第二次将其更改为 fully-buffered 时，缓冲区的大小变成了 1024(SIZE 值)，这正是程序想要的结果。再比较程序 7.1(buf_test.c)的执行结果，会发现标准输入的缓冲区类型的确发生了改变。

另外，细心的读者还会发现，在程序 7.2 中对标准输入流对象的缓冲区类型进行设置的时候，并没有首先打开这个流，而是直接进行设置。这是因为标准输入、标准输出和标准出错这 3 个流是在进程启动时自动打开的，并不需要用程序代码来打开它(参考 7.1.2 小节)。

2. 缓存的冲洗

所谓缓存的冲洗，是指将 I/O 操作写入缓存中的内容清空。这种清空可以是将流的内容完全丢掉，也可以是将其保存到文件中，相应的库函数如下：

```
#include <stdio.h>
int fflush(FILE *fp);
#include <stdio_ext.h>
void fpurge(FILE *fp);
```

fflush 函数用于将缓冲区中尚未写入文件的数据强制性地保存到文件中。调用成功时，返回值为 0；调用失败则返回 EOF。

fpurge 函数用于将缓冲区中的数据完全清除。由于使用较少，这个函数的定义在 <stdio_ext.h>中。

7.2 流的打开与关闭

当用户使用基于流的缓冲时会由 C 语言的库函数提供对缓冲的操作，用户则不用再耗费时间和精力控制缓冲区了。本节介绍基于缓冲区的文件 I/O 操作，这些函数都是标准 C 语言的函数，其他系统用户也可以使用其中的大多数函数。

要对流进行操作，首先要了解基于流的 I/O 操作中的最基本的库函数——流的打开与关闭的函数调用。

7.2.1　流的打开

对一个流进行操作之前，首先要将其打开，也就是建立某一个流同某个特定的文件或设备之间的关联，只有这样，才能对这个流进行各种操作。打开流的函数调用说明如下：

```
#include <stdio.h>
FILE *fopen (const char *pathname, const char *type);
FILE *freopen (const char *pathname, const char *type, FILE *fp);
FILE *fdopen (int fd, const char *type);
```

3 个函数的返回：若成功则返回文件指针，若出错则为 NULL(空指针)。

这 3 个函数的区别有以下几点。

- fopen 打开一个路径名由 pathname 指示的文件。
- freopen 在一个特定的流上(由 fp 指示)打开一个指定的文件(其路径名由 pathname 指示)，若该流已经打开，则先关闭该流。此函数一般用于将一个指定的文件打开为一个预定义的流：标准输入、标准输出或标准出错。
- fdopen 取一个现存的文件描述符(可能从 open、dup、dup2、fcntl 或 pipe 函数得到此文件描述符)，并使一个标准的 I/O 流与该描述符相结合。此函数常用于由创建管道和网络通信通道函数获得的描述符。因为这些特殊类型的文件不能使用标准 I/O 的 fopen 函数打开，首先必须先调用设备专用函数以获得一个文件描述符，然后用 fdopen 使一个标准 I/O 流与该描述符相结合。

说　明

fopen 和 freopen 是 ANSI C 的所属部分，而 ANSI C 并不涉及文件描述符，所以仅有 POSIX.1 标准具有 fdopen。

另外，在这 3 个函数中都含有 type 参数，它指定了对该 I/O 流的读、写方式，该值以一个字符串的形式传入。ANSI C 规定 type 参数可以有 15 种不同的值，它们列于表 7.2 中。

表 7.2　type 的取值及其对应模式

type 值	操作文件类型	是否新建文件	是否清空原文件	可读	可写	读写开始位置
r	文本文件	NO	NO	YES	NO	文件开头
r+	文本文件	YES	NO	YES	YES	文件开头
w	文本文件	YES	YES	NO	YES	文件开头
w+	文本文件	YES	YES	YES	YES	文件开头
a	文本文件	NO	YES	NO	YES	文件结尾
a+	文本文件	NO	YES	YES	YES	文件结尾
rb	二进制文件	NO	NO	YES	NO	文件开头

(续表)

type 值	操作文件类型	是否新建文件	是否清空原文件	可读	可写	读写开始位置
r+b 或 rb+	二进制文件	YES	NO	YES	YES	文件开头
wb	二进制文件	YES	YES	NO	YES	文件开头
w+b 或 wb+	二进制文件	YES	YES	YES	YES	文件开头
ab	二进制文件	NO	YES	NO	YES	文件结尾
a+b 或 ab+	二进制文件	NO	YES	YES	YES	文件结尾

说明

type 取值的字串中包含有字母“a”的表示“追加写”，即流打开以后，文件的读写位置在文件的末尾，所以成为追加写；type 字串中包括字母“b”的表示流以二进制文件的形式打开，其他的则表示流以文本文件的形式打开。不过需要说明的是，这一点对于 Linux 系统来讲是没有意义的，因为 Linux 系统下的二进制文件和文本文件都是普通文件，是字节流，内核并不区分两者。

如果成功打开流，fopen 函数将返回一个 FILE 对象的指针，用户可以使用该指针操作这个流；如果失败则返回 NULL，并且设置 errno 错误号。一般来讲，fopen 函数是很少出错的，出错的原因主要有以下 3 种：

- 指定的文件路径有误。
- type 参数是一个非法字符串。
- 文件的操作权限不够。

fdopen 函数用于在一个已经打开的文件描述符上打开一个流。与 fopen 有一点不同的是，由于文件已经被打开，所以 fdopen 函数不会创建文件，也不会将文件截短为 0，这一点要特别注意。这两步操作在打开该文件描述符的时候已经完成。

综上所述，当程序成功地完成了一个流的打开操作之后，它就和一个 FILE 结构指针联系起来了，随之进行的各种操作都是通过引用此结构指针进行库函数的调用来实现的。

7.2.2 流的关闭

在所需的操作完成以后，必须将流关闭。fclose 函数用于关闭一个流，其函数原型如下：

```
#include <stdio.h>
int fclose(FILE *fp);
```

返回：若成功则返回 0，若失败则返回 EOF。

说明

EOF 是一个定义在头文件<stdio.h>中的宏，其值是-1。

fclose 函数的参数 fp 是一个 FILE 对象的指针，它指向需要关闭的流。

需要注意的是，如果在程序结束前没有指向关闭流的操作，有可能会造成写入的数据停留

在缓冲区里，而没有保存到文件中，造成数据丢失。

程序 7.3 演示了打开和关闭一个流，代码如 stream.c 所示。

【程序 7.3】打开和关闭一个流：stream.c。

```
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    int fd;
    if( (fp = fopen("hello.txt", "r+")) == NULL)
    {   /*以读写方式打开流，若没有 hello.txt 文件，则创建它，从文件开头开始读写*/
        printf ("fail to open!\n");
        exit(1);                /*出错退出*/
    }
    fprintf(fp, "Hello! I like Linux C program!\n");
/*向该流输出一段信息，这段信息会保存到打开的文件上*/
    fclose(fp);                     /*操作完毕，关闭流*/
    if( (fd = open("hello.txt", O_RDWR)) == -1)
      {  /*以读写的方式打开文件——基于文件描述符的方式*/
         printf ("fail to open!\n");
         exit(1);            /*出错退出*/
      }
    if((fp = fdopen(fd, "a+")) == NULL)
      {  /*在打开的文件上打开一个流，并从文件尾开始读写*/
         printf ("fail to open stream!\n");
         exit(1);            /*出错退出*/
      }
    fprintf(fp, "I am doing Linux C programs!\n");
/*向该流输出一段信息，这段信息会保存到打开的文件上*/
    fclose(fp);                 /*关闭流，文件也被关闭*/
    return 0;
}
```

使用 gcc 编译 stream.c，并生成可执行文件 stream：

```
#gcc –o stream stream.c
```

运行程序之前先来查看一下 hello.txt 文件中的内容：

```
# cat hello.txt
```

文件内容为空。

运行程序，并再次查看文件 hello.txt 中的内容，得到输出结果：

```
# ./stream
# cat hello.txt
Hello! I like Linux C program!
I am doing Linux C programs!
```

从中可以看到，程序成功地将两条字符串语句写进了文件中。再次执行程序，并查看文件 hello.txt 中的内容，得到如下信息：

```
# ./stream
# cat hello.txt
Hello! I like Linux C program!
I am doing Linux C programs!
I am doing Linux C programs!
```

为什么第二次执行程序，hello.txt 文件中只要 3 条字符串语句呢？原来在代码 stream.c 中，字串“Hello! I like Linux C program!”的写操作是从文件的开头位置开始的(“r+”)，而字串“I am doing Linux C programs!”的写操作是从文件的结尾开始的(“a+”)，当第二次执行程序时，“Hello! I like Linux C program!”覆盖了原本位置的内容，而“I am doing Linux C programs!”是接在文件的末尾追加上去的。

另外，在程序 7.3 中使用了 fprintf 函数，它表示向指定的流中输出数据，在 7.3.4 小节中将向读者介绍。

7.2.3 流关闭前的工作

上面已经向读者提到，在对流的操作完毕后，必须关闭它。同时我们还知道，大多数流的操作都是基于缓冲机制的(注意 stderr 是无缓冲的)，那么就会出现一个问题：当关闭一个流时，那些缓存在缓冲区的、还未来得及写入文件或设备的数据怎么办呢？不必担心，fclose 替我们做好了这些工作。

在 fclose 函数关闭文件时,会将保存在内存中尚未来得及写回磁盘的文件内容写到磁盘上。实际上，fclose 函数在关闭流之前，还进行了最后一次写文件的操作。了解这一点很有必要，因为如果没有调用 fclose 函数，就必须等待内存中缓冲区被填满，由系统将其内容写回到磁盘上去。这也是在流操作完成之后必须关闭它的原因。

另外，对于 fclose 函数是否需要检查返回值的问题困扰着许多程序员。虽然严格上说，应该检查所有的系统调用的返回值，并且进行相应的出错处理，但是对于 fclose 函数还是很少有程序员检查其返回值。因为如果一个程序关闭的是本地文件，fclose 函数出错的几率很小，几乎为 0。但是需要注意的是，如果程序关闭的是一个网络环境中的远程文件，fclose 函数就有可能出错，此时就很有必要检查函数的返回值了。

由于 fclose 函数在关闭文件时会将缓冲区中的内容回写到磁盘上，因此 fclose 函数实际是附带进行了一个写文件的操作。我们知道，在网络环境中，文件的内容是要通过网络传输到目的主机上并且写入磁盘的，在这个传输过程中，如果网络连接出现问题或者传输数据出错，就会导致文件内容写入失败，这时 fclose 函数就会出错。

由此可知，如果在本地关闭一个文件，fclose 函数可以不检查返回值；如果在网络环境中关闭一个文件，检查 fclose 函数的返回值以判断文件是否关闭成功还是很有必要的。

7.3 流的读写

一旦打开了流，则可在 4 种不同类型的 I/O 中进行选择，来对其进行读、写操作。

- 基于字符的 I/O。每次读写一个字符数据的 I/O 方式，如果流是带缓存的，则由标准 I/O 函数处理所有缓存。
- 基于行的 I/O。当输入的内容遇到‘\n’换行符的时候，则将流中‘\n’之前的内容送到缓冲区中的 I/O 方式称为基于行的 I/O，即每次一行的 I/O。在进行相关函数调用时，应当说明能处理的最大行长。
- 直接 I/O。输入输出操作以记录为单位进行读写，即每次 I/O 操作读或写某种数量的对象，而每个对象具有指定的长度。
- 格式化 I/O。格式化输入输出是最常见的 I/O 方式，如 printf 和 scanf 函数。

说 明

直接 I/O(direct I/O)这个术语来自 ANSI C 标准，有时也被称为二进制 I/O、一次一个对象 I/O、面向记录的 I/O 或面向结构的 I/O。

7.3.1 基于字符的 I/O

基于字符的 I/O 通常是用来处理单个字符的，本小节向读者介绍字符的输入和字符的输出的函数调用。

1. 字符的输入

以下 3 个函数可用于一次读入一个字符：

```
#include <stdio.h>
int getc(FILE *fp);
int fgetc(FILE *fp);
int getchar(void);
```

3 个函数的返回：若成功则返回读入字符的值，若已处文件尾端或出错则为 EOF。

前两个函数中，参数 fp 表示所要读入字符的文件，它们的区别是 getc 可被实现为宏，而 fgetc 不能实现为宏。这意味着：

- getc 的参数不应当是具有副作用的表达式。

说 明

所谓副作用表达式，是相对于传统意义上的表达式来说的。例如，传统表达式的计算过程中，运算符不会令参与计算的变量本身的值发生改变；而 C/C++语言的表达式中由于++、--等运算符的介入，表达式求值可能导致参与计算的变量本身的值发生改变。这就是一种可能的副作用。关于副作用更深入的概念，并不属于本书的内容，读者可参考相关的 C/C++书籍。

➢ 因为 fgetc 一定是函数，所以可以得到其地址。这就允许将 fgetc 的地址作为一个参数传送给另一个函数。
➢ 调用 fgetc 所需时间很可能长于调用 getc，因为调用函数通常所需的时间长于调用宏。在<stdio.h>头文件中，getc 便是以宏定义的形式实现的，其编码具有较高的工作效率。

第三个函数 getchar 只能用来从标准输入流中输入数据，其作用相当于调用以 stdin 为参数的 getc 函数，即 getc(stdin)。

另外，这 3 个函数以 unsigned char 类型转换为 int 的方式返回下一个字符。说明为不带符号的理由是：即使最高位为 1 也不会使返回值为负。要求整型返回值的理由是：这样就可以返回所有可能的字符值再加上一个已发生错误或已到达文件尾端的指示值。在<stdio.h>中的常数 EOF 被要求是一个负值，其值经常是-1。这就意味着不能将这 3 个函数的返回值存放在一个字符变量中，以后还要将这些函数的返回值与常数 EOF 相比较。

还有一点需要注意，不管是出错还是到达文件尾端，这 3 个函数都返回同样的值。为了区分这两种不同的情况，必须调用 ferror 或 feof。它们的函数原型如下：

```
#include <stdio.h>
int ferror(FILE *fp);
int feof(FILE *fp);
```

两个函数返回：若条件为真则为非 0(真)，否则为 0(假)。

参数 fp 指向正在进行操作的文件。当读入字符出错时，ferror 条件为真；当位于文件尾时，feof 条件为真。

在大多数 UNIX 系统的 FILE 对象中，为每个流保持了两个标志：出错标志和文件结束标志。调用 clearerr 可以清除这两个标志：

```
#include <stdio.h>
void clearerr(FILE *fp);
```

从一个流读之后，可以调用 ungetc 将字符再送回流中。

```
#include <stdio.h>
int ungetc(int c, FILE *fp);
```

返回：若成功则返回要送回流中的字符的值，若出错则为 EOF。

ungetc 用于将读入的字符再推回流中，被推回的字符将在下一次读入操作时被读入，也可以在文件末尾推回某个字符，该字符也是将在下一次读入操作时被读入，读入字符的顺序与推回流中的顺序相反。参数 c 表示要推回的字符，被推回的字符并不一定要求是上一次读入的。文件的结束符 EOF 是不可被推回的。从原则上来讲，可以执行任意多次的推回操作来推回多个字符，但不提倡这样做。

当正在读一个输入流，并进行某种形式的分字或分记号操作时，会经常用到回送字符操作。有时需要先看一看下一个字符，以决定如何处理当前字符。然后就需要方便地将刚查看的字符送回，以便下一次调用 getc 时返回该字符。如果标准 I/O 库不提供回送能力，就需将该字符存放到一个我们自己的变量中，并设置一个标志以便判别在下一次需要一个字符时是调用 getc，还是从我们自己的变量中取用。

2. 字符的输出

以下 3 个函数用于字符的输出：

```
#include <stdio.h>
int putc(int c , FILE *fp);
int fputc(int c, FILE *fp);
int putchar(int c);
```

3 个函数返回：若成功则为 c，若出错则为 EOF。

putc 和 fputc 函数的第 1 个参数表示需要输出的字符，第 2 个参数表示输出的文件。如果成功输出一个字符，则返回输出的字符，如果出错则返回 EOF。

与输入函数一样，putchar(c)等同于 putc(c, stdout)，putc 可被实现为宏，而 fputc 则不能实现为宏。这里不再赘述。

程序 7.4 演示了使用每次输出一个字符的 I/O 方式实现一个类似于 cp 命令的简单的文件复制程序，在复制文件的同时在屏幕上输出该文件内容，代码清单如 char_copy.c。

【程序 7.4】基于字符 I/O 方式的文件复制：char_copy.c。

```
#include <stdio.h>
#include <stdlib.h>
#define Src_File "/home/zhangfan/CODE/hello.txt"
#define Des_File "/home/zhangfan/CODE/test.txt"
int main(void)
{
    FILE *fp1, *fp2;    /*源文件和目标文件的文件指针*/
    int c;              /*要进行输入和输出的字符*/
    if ((fp1 = fopen(Src_File, "rb"))==NULL)    /*以只读方式打开源文件 hello.txt */
{
        printf("fail to open source file\n");
        exit(1);        /*出错退出*/
    }
    if ((fp2 = fopen(Des_File, "wb"))==NULL)    /*以只写方式打开目标文件 test.txt*/
{
        printf ("fail to open des file\n");
        exit(1);
    }
    /*开始复制文件，每次读写一个字符*/
    while((c = fgetc(fp1))!=EOF)
{   /*读源文件，直到将文件内容全部读完*/
        if(fputc(c, fp2)==EOF)
          {   /*将读入的字符写到目标文件中去*/
            printf ("fail to write\n");
            exit(1);        /*出错退出*/
          }
       if(fputc(c, stdout)==EOF)
     {    /*将读入的字符输出到屏幕*/
            printf ("fail to write\n");
            exit(1);        /*出错退出*/
          }
```

```
        }
        fclose(fp1);    /*操作完毕，关闭源文件和目标文件*/
        fclose(fp2);
        return 0;
}
```

在编译运行程序之前，先来看看源文件 hello.txt 中的内容：

```
# cat hello.txt
Hello! I like Linux C program!
I am doing Linux C programs!
I am doing Linux C programs!
```

hello.txt 是笔者在执行程序 7.3 时生成的文件内容，在这个例子中我们就来使用这个文件(当然，读者也可以用别的文件)。

再来查看目标文件 test.txt 中的内容：

```
# cat test.txt
```

而此时 test.txt 文件中的内容是空的。

使用 gcc 编译 char_copy.c，并生成可执行文件 char_copy：

```
# gcc -o char_copy char_copy.c
```

运行程序，得到输出结果：

```
# ./char_copy
Hello! I like Linux C program!
I am doing Linux C programs!
I am doing Linux C programs!
```

从中可以看到，复制的数据已成功输出到屏幕上。此时再次查看 test.txt 文件的内容，输出信息如下：

```
# cat test.txt
Hello! I like Linux C program!
I am doing Linux C programs!
I am doing Linux C programs!
```

源文件 hello.txt 中的内容也成功复制到目标文件 test.txt 中去，得到了程序想要的结果。

7.3.2 基于行的 I/O

当输入的内容遇到‘\n’换行符的时候，则将流中‘\n’之前的内容送到缓冲区中的 I/O 方式称为基于行的 I/O，即每次一行的 I/O。下面介绍行的输入和输出。

1. 行的输入

fgets 和 gets 函数实现输入一行字符串，其函数原型如下：

```
#include <stdio.h>
char *fgets(char *buf, int n, FILE *fp);
```

```
char *gets(char *buf);
```

两个函数返回：若成功则返回缓冲区的首地址，若已处文件尾或出错则为 NULL。

fgets 函数的第 1 个参数表示的是存放读入串的缓冲区，第 2 个参数 n 表示读入的字符个数，此参数的最大值不能超过缓冲区的长度。fgets 函数一直读到遇到一个换行符‘\n’为止，如果在 n－1 个字符内未遇到换行符，则指读入 n－1 个字符。最后一个字节用于存储字符串结束标志‘\0’。需要注意的是 fgets 函数会将‘\n’换行符也读入到缓冲区中，也就是说缓冲区中的实际有效内容应该是缓冲区实际字节数(不包括‘\0’)减 1。fgets 函数的第 3 个参数是需要读入的流对象：

gets 函数和 fgets 函数类似，该函数从标准输入读取一行并将其存入一个缓冲区，并不将‘\n’读入到缓冲区中。gets 的返回值和 fgets 函数相同。

2. gets 函数的缺陷

gets 函数和 fgets 函数最大的不同是 gets 函数的缓冲区虽然由用户提供，但是用户无法指定其一次最多读入多少字节的内容，这一点导致 gets 变成了一个非常危险的函数。

程序 7.5 演示了 gets 函数的危险性，该程序定义了一个缓冲区，但是使用 gets 函数接收用户输入的字符串时却会出现问题。

【程序 7.5】gets 函数的危险性：gets_risk.c

```
#include <stdio.h>
int main(void)
{
    char buf[2048];          /*定义一个足够大的缓冲区*/
    while (gets(buf)==buf)
{   /*从屏幕读入一行字符串，当 gets 返回值不为缓冲区的首地址时则停止读入*/
        printf("%s\n", buf);   /*并且将该字符串显示输出到屏幕上*/
    }
    return 0;
}
```

使用 gcc 编译 gets_risk.c，并生成可执行文件 gets_risk：

```
# gcc -o gets_risk gets_risk.c
```

运行程序，得到输出结果：

```
# ./gets_risk
Hello, Linux world ↙(↙表示回车)
Hello, Linux world
welcome to the Linux world, it is painful, but you will love it ↙(↙表示回车)
welcome to the Linux world, it is painful, but you will love it
```

到目前为止都没有出现问题，事实上，在命令行终端的情况下不会出现问题。因为 Shell 终端的输入缓冲区只有 1024 个字节，也就是说我们的攻击实际上被 Shell 挡住了。

这个时候换一种方式，先使用 Ctrl+c 结束该程序，然后将一个非常大的文件重定向到标准输入上。方法如下，bigfile.txt 是我们假设的在当前目录下存在的一个内容很多(远大于 2048 字节)的文本文件：

```
# ./gets_risk < bigfile.txt
segmentation fault
```

此时系统产生了错误信息：段错误。原因就是输入的字符过多，造成了 gets 函数的缓冲区越界，而导致程序崩溃了。由此可见，gets 函数确实是一个非常不安全的函数，所以笔者不推荐读者使用该函数。

说 明

在第 6 章中，多次在程序中用到了 gets 函数。并且在 6.2.4 小节我们看到了那条警告信息：

```
warning: In function `main':
: the `gets' function is dangerous and should not be used.
```

笔者在前面使用 gets 并不是要求读者学着那样做，恰恰相反，是通过实例来警戒读者，从开始编程的时候就应该养成一种好习惯，尽量规避那些具有缺陷和漏洞的 C 函数。《C 缺陷与陷阱》是关于这方面的一本不错的书籍。

3. 行的输出

fputs 和 puts 函数实现输出一行字符串，其函数原型如下：

```
#include <stdio.h>
int fputs(const char *buf, FILE *restrict fp);
int puts(const char * str);
```

fputs 函数的第 1 个参数表示存放输出内容的缓冲区，第 2 个参数表示要输出的文件。如果成功输出，则返回输出的字节数，失败则返回-1。

puts 函数用于向标准输出输出一行字符串，其参数和 fputs 函数的第 1 个参数相同，如果成功输出，则返回输出字节数，失败则返回-1。

fputs 函数和 puts 函数都不输出字符串的结束符‘\0’。puts 函数不像 gets 函数那样有致命的漏洞，但是其安全程度也不是很高，所以笔者依然推荐读者使用 fputs 函数。对于行 I/O 来说，fgets 函数和 fputs 函数的搭配是安全又可靠的。

7.3.3 直接 I/O

在 7.3.1 小节和 7.3.2 小节中介绍的函数是以一次一个字符或一次一行的方式进行 I/O 操作的。比如需要一次读或写整个结构，为了使用 getc 或 putc 做到这一点，必须循环通过整个结构，一次读或写一个字节。因为 fputs 在遇到 NULL 字节时就停止，而在结构中很可能含有 NULL 字节，所以不能使用每次一行函数实现这种要求。与此类似，如果输入数据中包含有 NULL 字节或新行符‘\n’，则 fgets 也不能正确工作。因此，提供了下列两个函数以执行二进制 I/O 操作，也称直接 I/O 操作。函数原型如下：

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *fp);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *fp);
```

两个函数的返回：读或写的对象数。

fread 函数用于执行直接输出操作。参数 ptr 是指向读取数据的缓冲区的指针，size 是读取对象的大小，nmemb 表示欲读取的对象的个数，fp 是指向要读取的流的 FILE 结构指针。

fwrite 函数用于执行直接输入操作。参数 ptr 是指向存放将要输入数据的缓冲区的指针，size 是写入对象的大小，nmemb 表示欲写入的对象的个数，fp 是指向要写入的流的 FILE 结构指针。

这些函数有两个常见的用法：

(1) 读或写一个二进制数组。例如，将一个浮点型数组的第 2~5 个元素写到一个文件上，可以这样编写代码：

```
float data[10];
if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
printf("fwrite error!\n");
```

其中，指定 size 为每个数组元素的长度，nmemb 为欲写的元素数。

(2) 读或写一个结构。例如，可以这样写：

```
struct
{
short count;
long total;
char name[NAME_SIZE];
} item;
if (fwrite(&item, sizeof(item), 1, fp) != 1)
printf ("fwrite error!\n");
```

其中，指定 size 为结构的长度，nmemb 为 1(要写的对象数)。

将这两个例子结合起来就可读或写一个结构数组。为了做到这一点，size 应当是该结构的 sizeof，nmemb 应是该数组中的元素数。

fread 和 fwrite 返回读或写的对象数。对于读，如果出错或到达文件尾端，则此数字可以少于 nmemb。在这种情况下，应调用 ferror 或 feof 以判断究竟是哪一种情况(见 7.3.1 小节)。对于写，如果返回值少于所要求的 nmemb，则出错。

程序 7.6 演示了如何使用 fwrite 函数和 fread 函数实现简单的文件复制。该程序使用 fread 函数从文件中读出内容，之后使用 fwrite 函数输出到另一个文件中。代码如 direct_copy.c 所示。实例中还是使用了笔者惯用的例子文件 hello.txt。

【程序 7.6】基于直接 I/O 方式的文件复制：direct_copy.c。

```
#include <stdio.h>
#include <stdlib.h>
#define Src_File "/home/zhangfan/CODE/hello.txt"
#define Des_File "/home/zhangfan/CODE/test.txt"
int main(void)
{
    FILE *fp1, *fp2;        /*源文件和目标文件的文件指针*/
    char buf[1024];
    int n;
if ((fp1=fopen(Src_File, "rb"))==NULL)
```

```
/*以只读方式打开源文件，读开始位置为文件开头*/
{
            printf("fail to open source file\n");
            exit(1);          /*出错退出*/
        }
      if ((fp2=fopen(Des_File, "wb"))==NULL)
/*以只写方式打开目标文件，写开始位置为文件结尾*/
{
            printf("fail to open des file\n");
            exit(1);          /*出错退出*/
        }
/*开始复制文件，文件可能很大，缓冲一次装不下，所以使用一个循环进行读写*/
      while ((n=fread(buf, sizeof(char), 1024, fp1)) > 0)
{   /*读源文件，直到将文件内容全部读完*/
            if (fwrite(buf, sizeof(char), n, fp2)==-1)
             {   /*将读出的内容全部写到目标文件中去*/
                printf("fail to write\n");
                exit(1);       /*出错退出*/
             }
        }
      if(n==-1)
{   /*如果因为读入字节小于 0 而跳出循环，则说明出错了*/
            printf("fail to read\n");
            exit(1);        /*出错退出*/
        }
      fclose(fp1);     /*操作完毕，关闭源文件和目标文件*/
      fclose(fp2);
      return 0;
}
```

在编译运行程序之前，先来看看源文件 hello.txt 中的内容：

```
# cat hello.txt
Hello! I like Linux C program!
I am doing Linux C programs!
I am doing Linux C programs!
```

再来查看目标文件 test.txt 中的内容：

```
# cat test.txt
Hello! I like Linux C program!
I am doing Linux C programs!
I am doing Linux C programs!
```

使用 gcc 编译 direct_copy.c，并生成可执行文件 direct_copy：

```
#gcc –o direct_copy direct_copy.c
```

运行程序，并查看文件 test.txt 中的内容，得到输出结果：

```
# ./direct_copy
# cat test.txt
```

```
Hello! I like Linux C program!
I am doing Linux C programs!
I am doing Linux C programs!
Hello! I like Linux C program!
I am doing Linux C programs!
I am doing Linux C programs!
```

从中可以看到，hello.txt 文件中的内容以追加的方式复制到目标文件 test.txt 中去了。

另外，使用直接 I/O 的一个基本问题是它只能用于读或写在同一系统上的数据。在多年之前，这并无问题(那时，所有 UNIX 系统都运行于 PDP-11 上)，而现在，很多异构系统通过网络相互连接起来，而且，这种情况已经非常普遍。常常有这种情形，在一个系统上写的数据，在另一个系统上处理。在这种环境下，这两个函数可能就不能正常工作，其原因是：

(1) 在一个结构中，同一成员的位移量可能随编译程序和系统的不同而异(由于不同的对准要求)。某些编译程序有一选择项，它允许紧密包装结构(节省存储空间，而运行性能则可能有所下降)或准确对齐，以便在运行时易于存取结构中的各成员。这意味着即使在单一系统上，一个结构的二进制存放方式也可能因编译程序的选择项而不同。

(2) 用来存储多字节整数和浮点值的二进制格式在不同的系统结构间也可能不同。

7.3.4 格式化 I/O

基于流的 I/O 操作的一个最大的特点就是它可以实现格式化的输入输出(基于字符和行的 I/O，以及直接 I/O 统称为非格式化 I/O)。格式化 I/O 是最常见的 I/O 方式，比如我们常用的 printf 和 scanf。

1. 格式化输出

执行格式化输出处理的是 4 个 printf 函数：

```
#include <stdio.h>
int printf(const char *format, ...);
int fprintf(FILE *fp, const char *format, ...);
```

两个函数的返回：若成功则为实际输出的字符数，若输出出错则为负值。

```
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
```

两个函数的返回：若成功则为实际存入缓冲区的字符数，若出错则为负值。

- printf 是读者已经很熟悉的一个函数了，它用于向标准输出流中输出数据，参数 format 是一个字符串指针，用于描述输出格式(参考 2.6.3 小节)。第二个参数写为...，这是 ANSI C 说明余下参数的数目和类型可以变化的方法。
- fprintf 函数用于向指定的流中输出数据，参数 fp 指向要进行输出的流。
- sprintf 函数用于向指定的流中输出一个字符串，参数 str 是字符串指针，指向要进行输出的缓冲区。sprintf 字符串的尾端自动添加一个 NULL 字节，但该字节不包括在返回值中。

➢ snprintf 函数的作用同 sprintf 相似。不同的是 sprintf 函数不能对缓冲区进行处理，使用时可能会出现缓冲区的溢出而导致程序崩溃，而 snprintf 可以处理缓冲区，参数 size 用于设定缓冲区的大小。

注 意

在一些其他操作系统的 stdio 库中，并不提供 snprintf 函数，而 Linux 提供 snprintf。

2. 格式化输入

执行格式化输入处理的是 3 个 scanf 函数。

```
#include <stdio.h>
int scanf(const char *format, ...);
int fscanf(FILE *fp, const char *format, ...);
int sscanf(char *str, const char *format, ...);
```

3 个函数的返回：若成功则返回指定的输入项数，若输入出错，或在任意变换前已至文件尾端则为 EOF。

这 3 个格式化输入函数的说明同 printf 族函数十分类似，简要说明如下：

➢ scanf 函数用于从标准输入流中输入数据。
➢ fscanf 函数用于从一个指定的流中输入数据，参数 fp 指定该流。
➢ sscanf 函数用于从一个字符串中输入数据，参数 str 指定该字符串。

注 意

由于从缓冲区读入数据不存在造成缓冲溢出的情况，stdio 库中没有与 snprintf 相应的格式化输入函数。

程序 7.7 使用 fprintf 函数和 fscanf 函数实现了向指定的文件流中输出和输入数据。程序中先使用 fprintf 函数将 buf 数组中的字符串写入指定的文件，再使用 fscanf 函数读取文件中的内容到数组 buf2 中，最后将 buf2 中的内容打印出来。代码如 format_io.c。

【程序 7.7】使用 fprintf 和 fscanf 函数实现输出和输入：format_io.c。

```
#include <stdio.h>
#define File_path "/home/zhangfan/CODE/hello.txt"
int main(void)
{
FILE *fp;          /*指向操作文件的文件指针*/
char buf[ ]="Hello! I like Linux C program!";
char buf2[80];
fp=fopen(File_path,"w");     /*以只写方式打开文件*/
fprintf(fp, "%s", buf);      /*向该文件流输出字符串数据 buf*/
fprintf(fp, "\n");           /*向该文件流输出换行*/
fclose(fp);                  /*关闭文件流*/
fp=fopen(File_path, "r");    /*再以只读方式打开文件*/
fscanf(fp,"%s",&buf2);       /*将该文件流中的数据读入到 buf2*/
fclose(fp);
printf("%s\n",buf2);         /*打印 buf2 的内容*/
```

```
return 0;
}
```

在编译运行程序之前，先来查看 hello.txt 文件中的内容：

```
# cat hello.txt
```

此时 hello.txt 文件中的内容为空。

使用 gcc 编译 format_io.c，并生成可执行文件 format_io：

```
#gcc –o format_io format_io.c
```

运行程序，得到输出结果：

```
# ./format_io
Hello!
```

再次查看 hello.txt 文件中的内容：

```
# cat hello.txt
Hello! I like Linux C program!
```

从中可以看到，buf 数组中的字符串已成功写入到指定的 hello.txt 文件中。另外，细心的读者还会发现，程序的最后打印 buf2 数组中的内容的结果是“Hello!”，而不是全部的字符串“Hello! I like Linux C program!”，这是因为在使用 fscanf 函数读取文件流的内容时，fscanf 在遇到空格符、制表符和回车符时会自动中止读取字符。也是就说，在程序 7.7 中，数组 buf2 中的内容实际上是“Hello!”，而不是“Hello! I like Linux C program!”。

7.4 本章小结

本章介绍了基于流的输入输出的有关概念和操作。基于流的 I/O 操作是由标准 C 函数库提供的，与基于文件描述符的 I/O 操作相比，基于流的 I/O 更简单、方便。在大多数情况下，程序员更愿意使用基于流的输入输出方法。本章详细介绍了各种不同的输入输出方法，对于实际的应用程序的编写十分有用，读者应该熟练掌握它们。

实战演练

1. 编写一个程序，查看本机中标准输入流的缓冲区类型，体会流对象中的缓冲区标志与缓冲区类型宏的逻辑关系。

2. 编写一个程序，使用 fopen 和 fclose 函数打开与关闭/root 目录下的 install.log 文件，并返回函数的执行结果。

3. 编写一个程序，使用基于字符 I/O 的方式将/home/zhangfan/hello 文件中的内容读取到/home/zhangfan/tmp 文件中。

4. 编写一个程序，使用基于直接 I/O 的方式向/home/zhangfan/tmp 文件中写入字符串数据

“Linux C program.”。

5. 编写一个程序，使用基于格式化 I/O 的方式向/home/zhangfan/tmp 文件中写入字符串数据“Linux C program.”。

6. 编写一个程序，使用基于直接 I/O 的方式读取/root 目录下 install.log.syslog 文件中的内容到终端输出。

7. 编写一个程序，使用基于格式化 I/O 的方式读取/root 目录下 install.log.syslog 文件中的内容到终端输出。

第8章

进 程 控 制

进程是操作系统中一个非常重要的概念，它是一个程序的一次执行过程，程序是进程的一种静态描述，系统中运行的每一个程序都是在它的进程中运行的。了解 Linux 系统中进程的概念和属性，熟悉使用进程的操作和进程控制的相关系统调用，会使用户在使用 Linux 系统完成各种工作时更加得心应手。



本章内容：

◎ 进程的基本概念。

◎ Linux 下进程控制的相关函数调用。

◎ 多个进程之间的关系。

8.1 进程的基本概念

进程的概念起源于 20 世纪 60 年代，目前已成为各种操作系统和并发程序设计中非常重要的概念。可以说，用户在操作系统中所做的每一件事，都是通过进程实现的。要灵活使用进程，首先需要了解进程的一些基本概念。

8.1.1 Linux 进程简介

Linux 是一个多用户多任务的操作系统，多用户是指多个用户可以在同一时间使用同一台计算机系统；多任务是指 Linux 可以同时执行几个任务，它可以在还未执行完一个任务时又执行另一项任务，操作系统管理着多个用户的请求和多个任务。

大多数系统都只有一个 CPU 和一个主存，但一个系统可能有多个二级存储磁盘和多个输入/输出设备。操作系统管理这些资源并在多个用户间共享资源，当用户提出一个请求时，给用户造成一种假象，好像系统只被用户独自占用。而实际上操作系统监控着一个等待执行的任务队列，这些任务包括用户作业、操作系统任务、邮件和打印作业等。操作系统根据每个任务的优先级为每个任务分配合适的时间片，每个时间片大约都有零点几秒，虽然看起来很短，但实际上已经足够计算机完成成千上万的指令集。每个任务都会被系统运行一段时间，然后挂起，系统转而处理其他任务；过一段时间以后再回来处理这个任务，直到某个任务完成，从任务队列中去除。

Linux 系统中所有运行的东西都可以称之为一个进程。每个用户任务、每个系统管理，都可以称之为进程，Linux 用分时管理方法使所有的任务共同分享系统资源。我们讨论进程的时候，不会去关心这些进程究竟是如何分配的，或者是内核如何管理分配时间片的，我们所关心的是如何去控制这些进程，让它们能够很好地为用户服务。

进程的一个比较正式的定义是：在自身的虚拟地址空间运行的一个单独的程序。进程与程序是有区别的，进程是动态的，程序是静态的，进程不是程序，虽然它由程序产生。程序只是一个静态的命令集合，不占系统的运行资源；而进程是一个随时都可能发生变化的、动态的、使用系统运行资源的程序，而且一个程序可以启动多个进程。

提 示

我们在理解进程和程序的关系时，可以把它们想象成是戏剧和剧本的关系。一个程序可以对应多个进程，就像一个剧本可以拿来拍摄多部电视剧；但一个进程只能对应一个程序，就像一部电视剧只能使用一个剧本。剧本是静态的，而戏剧是动态的。

归结起来，进程具有以下 4 个要素：

- 要有一段程序供该进程运行。
- 进程专用的系统堆栈空间。
- 进程控制块(稍后会介绍)，在 Linux 中的具体实现是 task_struct 结构。
- 有独立的存储空间。

当进程缺少四要素中的一个要素时，我们称其为线程。

Linux 系统中所有进程都是相互联系的，除了初始化进程外，所有进程都有一个父进程。新进程不是被创建，而是被复制，或者从以前的进程复制而来。Linux 系统中所有的进程都是由一个进程号为 1 的 init 进程衍生而来的，而我们在 Shell 下执行程序启动的进程则是 Shell 进程的子进程，当然我们启动的进程可以再启动自己的子进程。这样形成了一棵进程树，每个进程都是树中的一个节点，其中树的根是 init。

Linux 操作系统包括 3 种不同类型的进程，每种进程都有自己的特点和属性：

- 交互进程：由一个 Shell 启动的进程。交互进程既可以在前台运行，也可以在后台运行。
- 批处理进程：这种进程和终端没有联系，是一个进程序列。
- 监控进程(也称守护进程)：Linux 系统启动时启动的进程，并在后台运行。

8.1.2 进程与作业

作业(task)也是操作系统中一个很常见的概念，作业和进程一样，都是操作系统正在执行的程序，但作业不是进程，进程和作业的概念是有区别的。

一个正在执行的进程称为一个作业，而作业可以包含一个或多个进程，尤其是当使用了管道和重定向命令的时候。例如“nroff -man ps.1 | grep kill | more”这个作业就同时启动了 3 个进程。

作业控制指的是控制正在运行的进程的行为。比如，用户可以挂起一个进程，等一会儿再继续执行该进程。Shell 将记录所有启动的进程情况，在每个进程过程中，用户可以任意挂起进程或重新启动进程。作业控制是许多 Shell(包括 bash 和 tcsh)的一个特性，使用户能在多个独立作业间进行切换。

一般而言，进程与作业控制相关联时，才被称为作业。

例如，当用户编辑一个文字文件，并需要中止做其他事情时，利用作业控制，用户可以让编辑器暂时挂起，返回 Shell 提示符开始做其他的事情。其他事情做完以后，用户可以重新启动挂起的编辑器，返回到刚才中止的地方，就像用户从来没有离开编辑器一样。这只是一个例子，作业控制还有许多其他实际的用途。

8.1.3 进程标识

在 Linux 中，每个进程在创建时都会被分配一个数据结构，称为进程控制块(Process Control Block，简称 PCB)。PCB 是系统为了管理进程设置的一个专门的数据结构，用它来记录进程的外部特征，描述进程的运动变化过程。系统利用 PCB 来控制和管理进程，所以 PCB 是系统感知进程存在的唯一标志。进程与 PCB 是一一对应的关系。

在不同的操作系统中对进程的控制和管理机制不同，PCB 中的信息多少也不一样，通常 PCB 应包含如下一些信息：

- 进程标识符。每个进程都必须有一个唯一的标识符，可以是字符串，也可以是一个数字。
- 进程当前状态。用来说明进程当前所处的状态。为了管理的方便，系统设计时会将相同的状态的进程组成一个队列，如就绪进程队列，等待进程则要根据等待的事件组成多个等待队列，如等待打印机队列、等待磁盘 I/O 完成队列等。
- 进程相应的程序和数据地址，以便把 PCB 与其程序和数据联系起来。

- 进程资源清单。列出所拥有的除 CPU 外的资源记录，如拥有的 I/O 设备，打开的文件列表等。
- 进程优先级。进程的优先级反映进程的紧迫程度，通常由用户指定和系统设置。
- CPU 现场保护区。当进程因某种原因不能继续占用 CPU 时(如等待打印机)，释放 CPU，这时就要将 CPU 的各种状态信息保护起来，为将来再次得到处理机恢复 CPU 的各种状态，继续运行。
- 进程同步与通信机制。用于实现进程间互斥、同步和通信所需的信号量等。
- 进程所在队列 PCB 的链接字。根据进程所处的现行状态，进程相应的 PCB 参加到不同队列中。PCB 链接字指出该进程所在队列中下一个进程 PCB 的首地址。
- 与进程有关的其他信息。如进程记账信息，进程占用 CPU 的时间等。

PCB 中包含了很多重要的信息，供系统调度和进程本身执行使用，其中最重要的莫过于进程 ID(process ID)了，进程 ID 也被称为进程标识符，是一个非负的整数，在 Linux 操作系统中唯一地标识一个进程，在我们最常使用的 I386 架构(即 PC 使用的架构)上，一个非负的整数的变化范围是 0～32767，这也是我们所有可能取到的进程 ID。其实从进程 ID 的名字就可以看出，它就是进程的“身份证号码”，就像每个人的身份证号码都不会相同，每个进程的进程 ID 也不会相同。

在 Linux 系统中有某些专用的进程：进程 ID 0 是调度进程，常常被称为交换进程(swapper)。该进程并不执行任何磁盘上的程序——它是内核的一部分，因此也被称为系统进程。进程 ID 1 通常是 init 进程，在自举过程结束时由内核调用。该进程的程序文件在 UNIX 的早期版本中是 /etc/init，在较新版本中是/sbin/init。此进程负责在内核自举后起动一个 UNIX 系统。init 通常读与系统有关的初始化文件(/etc/rc*文件)，并将系统引导到一个状态(例如多用户)。init 进程决不会终止。它是一个普通的用户进程(与交换进程不同，它不是内核中的系统进程)，但是它以超级用户特权运行。

一个或多个进程可以合起来构成一个进程组(process group)，一个或多个进程组可以合起来构成一个会话(session)。这样我们就有了对进程进行批量操作的能力，比如通过向某个进程组发送信号来实现向该组中的每个进程发送信号。

提 示

可以通过“ps”命令察看自己的系统中目前有多少进程正在运行，使用“ps-aux”命令还会列出当前系统中每一个进程的详细信息。读者不妨自己试一试。

那么，如何获取系统中的某一进程的标识呢？Linux 提供了 getpid 函数调用来返回系统中当前进程的进程 ID，函数原型如下：

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
```

函数返回：调用进程的进程 ID。

getpid 的使用很简单，就是返回当前进程的进程 ID，一个简单的示例如程序 8.1。

【程序 8.1】获得当前进程的进程 ID：get_ pid.c。

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(void)
{
   printf("The current process ID is %d\n",getpid());
   return 0;
}
```

细心的读者可能注意到了，在程序中使用了“%d”将 pid_t 类型的返回值(pid_t 类型即为进程 ID 的数据类型)打印出来，难道 pid_t 类型的数据和 int 类型是完全兼容的吗？答案的确如此。事实上，在 i386 架构上(就是一般 PC 计算机的架构)，pid_t 类型是和 int 类型完全兼容的，我们可以用处理整形数的方法去处理 pid_t 类型的数据。所以，在 get_pid.c 中，省去“#include <sys/types.h>”这行代码是行得通的。

使用 gcc 编译 get_pid.c，并生成可执行文件 get_pid：

```
#gcc –o get_pid get_pid.c
```

运行程序，得到输出结果：

```
#./get_pid
The current process ID is 2033
```

当然，在不同的主机上运行这个程序，结果(getpid()的返回值)可能是完全不同的，这是很正常的。再运行一次：

```
#./get_pid
The current process ID is 2034
```

可以看到，尽管是同一系统下的同一个应用程序，每一次运行的时候，所分配的进程标识符都是不相同的。

8.2 进程控制的相关函数

进程的控制是通过函数调用来实现的，其中包括系统调用函数，也包括库函数。本节向读者介绍进程的创建、等待、终止等具体操作的相关函数调用。

8.2.1 fork 和 vfork 函数

要创建一个进程，最基本的系统调用是 fork。系统调用 fork 用于派生一个进程，函数原型如下：

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

返回：若成功，父进程中返回子进程 ID，子进程中返回 0；若出错则返回-1。

fork 系统调用的作用是复制一个进程。当一个进程调用它，完成后就出现两个几乎一模一样的进程，我们也由此得到了一个新进程。由 fork 创建的新进程被称为子进程(child process)，而将原来的进程称为父进程(parent process)。子进程是父进程的一个复制，即子进程从父进程得到了数据段和堆栈段的复制，这些需要分配新的内存；而对于只读的代码段，通常使用共享内存的方式访问。

fork 函数的主要用途：

- 一个进程希望复制自身，从而父子进程能同时执行不同段的代码。
- 进程想执行另外一个程序。

fork 函数被调用一次，但返回两次(这样的函数在 Linux 中只有少数几个)。两次返回的区别是子进程的返回值是 0，而父进程的返回值则是新子进程的进程 ID。将子进程 ID 返回给父进程的理由是：因为一个进程的子进程可以多于一个，所以没有一个函数使一个进程可以获得其所有子进程的进程 ID。fork 使子进程得到返回值 0 的理由是：一个进程只会有一个父进程，所以子进程总是可以调用 getppid 以获得其父进程的进程 ID(进程 ID 0 总是由交换进程使用，所以一个子进程的进程 ID 不可能为 0)。

fork 返回后，子进程和父进程都从调用 fork 函数的下一条语句开始执行。

说 明

fork 的中文含义是“叉，叉形物”，fork 函数的名字就是来源于这个与叉子的形状颇有几分相似的工作流程。

一般来说，在 fork 之后是父进程先执行还是子进程先执行是不确定的。这取决于内核所使用的调度算法。如果要求父、子进程之间相互同步，则要求某种形式的进程间通信。关于进程间通信，将在第 10 章讲到。

另外，现在很多的 Linux 系统实现中，并不是做一个父进程数据段和堆栈的完全复制，因为在 fork 之后经常跟随着 exec(见 8.2.2 小节)。作为替代，使用了在写时复制(Copy-On-Write，COW)的技术。这些区域由父、子进程共享，而且内核将它们的存取许可权改变为只读的。如果有进程试图修改这些区域，则内核为有关部分，典型的是虚存系统中的“页”，做一个复制。

在 Linux 中，创造新进程的方法只有一个，就是我们正在介绍的 fork。其他一些库函数，如 system()，看起来似乎它们也能创建新的进程，如果阅读一下它们的源代码就会明白，它们实际上也在内部调用了 fork。包括我们在命令行下运行应用程序，新的进程也是由 Shell 调用 fork 制造出来的。fork 有一些很有意思的特征，下面通过一个小程序来对它有更多的了解。代码如程序 8.2 所示。

【程序 8.2】fork 函数的使用：fork_test.c。

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main (void)
{
    int count=0;
```

```
pid_t pid;              /*此时仅有一个进程*/
    pid=fork();         /*此时已经有两个进程在同时运行*/
    if(pid < 0)
      {
          printf("error in fork!");
          exit(1);      /* fork 出错退出*/
      }
    else if(pid==0)
      printf("I am the child process, the count is %d, my process ID is %d\n",count,getpid());
    else
      printf("I am the parent process, the count is %d, my process ID is %d\n",++count,getpid());
    return 0;
}
```

使用 gcc 编译 fork_test.c，并生成可执行文件 fork_test：

```
#gcc –o fork_test fork_test.c
```

运行程序，得到输出结果：

```
#./fork_test
I am the parent process, the count is 1, my process ID is 2053
I am the child process, the count is 0, my process ID is 2054
```

读者在阅读这个程序的时候，必须首先了解一个概念：在语句 pid=fork();之前，只有一个进程在执行这段代码，但在这条语句之后，就变成两个进程在执行了，这两个进程的代码部分完全相同，将要执行的下一条语句都是 if(pid < 0)……。

两个进程中，原先就存在的那个被称为“父进程”，新出现的那个被称作“子进程”。父子进程的区别除了进程标志符(process ID)不同外，变量 pid 的值也不相同，pid 存放的是 fork 的返回值。fork 调用的一个奇妙之处就是它仅仅被调用一次，却能够返回两次，它可能有 3 种不同的返回值：

- 在父进程中，fork 返回新创建子进程的进程 ID。
- 在子进程中，fork 返回 0。
- 如果出现错误，fork 返回一个负值。

fork 出错可能有两种原因：一是当前的进程数已经达到了系统规定的上限，这时 errno 的值被设置为 EAGAIN；二是系统内存不足，这时 errno 的值被设置为 ENOMEM。

说 明

errno 是 Linux 下的一个宏定义常量，它被定义成不同的正整数。当 Linux 中的 C API 函数发生异常时，一般会将 errno 变量(需 include<errno.h>)赋一个整数值，不同的值表示不同的含义，可以通过查看该值推测出错的原因，在实际编程中使用 errno 可以解决不少原本看来莫名其妙的问题。

读者可以在/usr/include/asm/errno.h 文件中看到 errno 常量的定义。

fork 系统调用出错的可能性很小，而且如果出错，一般都为第一种错误。如果出现第二种错误，说明系统已经没有可分配的内存，正处于崩溃的边缘，这种情况对 Linux 来说是很罕见的。

讲到这里，读者可能已经完全看懂剩下的代码了：如果 pid 小于 0，说明出现了错误；如果 pid 等于 0，就说明 fork 返回了 0，也就说明当前进程是子进程，就去执行 printf("I am the child process ")，并输出 count 值 0；否则(else)，当前进程就是父进程，执行 printf("I am the parent process ")，并输出++count 值 1。在 fork 中，父子进程是独立开来的，并没有影响。

Linux 下，另一个创建进程的方法是 vfork，它的函数原型如下：

```
#include <sys/types.h>
#include <unistd.h>
pid_t vfork(void);
```

返回：与 fork 相同，父进程中返回子进程的进程号，在子进程中返回 0，若出错则返回-1。

fork 与 vfork 之间是有区别的。fork 要复制父进程的数据段；而 vfork 则不需要完全复制父进程的数据段，在子进程没有调用 exec 或 exit(稍后会讲解这两个函数)之前，子进程与父进程共享数据段。fork 不对父子进程的执行次序进行任何限制；而在 vfork 调用中，子进程先运行，父进程挂起，直到子进程调用了 exec 或 exit 之后，父子进程的执行次序才不再有限制。

事实上，vfork 创建出来的并不是真正意义上的进程，而是一个线程，因为它缺少了进程的四要素的第 4 项——独立的内存资源(参考 8.1.1 小节)。

程序8.3 说明了 vfork 创建的子进程与父进程之间是共享数据段的。代码清单如vfork_test1.c 所示。

【程序 8.3】vfork 创建的子进程与父进程之间共享数据段：vfork_test1.c。

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main (void)
{
int count=1;
    int child;
    printf("Before create son, the father's count is:%d\n", count);    /*创建进程之前*/
    child = vfork();          /*此时已经有两个进程在同时运行*/
    if(child < 0)
      {
         printf("error in vfork!");
         exit(1);        /* fork 出错退出*/
       }
    if(child==0)        /*子进程*/
      {
         printf("This is son, his pid is: %d and the count is: %d\n", getpid(), ++count);
         exit(1);
      }
else                    /*父进程*/
      {
         printf("After son, This is father, his pid is: %d and the count is: %d, and the child is: %d\n", getpid(),
count, child);
      }
```

```
return 0;
}
```

使用 gcc 编译 vfork_test1.c，并生成可执行文件 vfork_test1：

```
#gcc -o vfork_test1 vfork_test1.c
```

运行程序，得到输出结果：

```
#./vfork_test1
Before create son, the father's count is:1
This is son, his pid is: 2611 and the count is: 2
After son, This is father, his pid is: 2610 and the count is: 2, and the child is: 2611
```

从代码执行的结果可以看到，在子进程中修改了 count 值(执行++count，其值变为 2)，但是在父进程中输出 count 值时也是 2，这说明子进程和父进程是共享 count 的，也就是说，由 vfork 创建出来的子进程与父进程之间是共享内存区的。

另外，由 vfork 创造出来的子进程还会导致父进程挂起，除非子进程执行了 exit 或者 execve 才会唤起父进程，程序 8.4 演示了这个问题。代码如 vfork_test2.c 所示。

【程序 8.4】vfork 创建的子进程导致父进程挂起：vfork_test2.c。

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main (void)
{
int count = 1;
    int child;
    printf("Before create son, the father's count is:%d\n", count);     /*创建进程之前*/
    if(!(child = vfork()))
      {  /*这里是子进程执行区*/
         int i;
         for(i = 0; i < 100; i++)
           {
printf("This is son, The i is: %d\n", i);
              if(i == 70)
                 exit(1);
           }
          printf("This is son, his pid is: %d and the count is: %d\n", getpid(), ++count);
          exit(1);   /*子进程退出*/
      }
     else
      {  /*父进程执行区*/
          printf("After son, This is father, his pid is: %d and the count is: %d, and the child is: %d\n", getpid(),
count, child);
      }
return 0;
}
```

使用 gcc 编译 vfork_test2.c，并生成可执行文件 vfork_test2：

```
#gcc -o vfork_test2 vfork_test2.c
```

运行程序，得到输出结果：

```
#./vfork_test2
Before create son, the father's count is:1
This is son, The i is: 0
This is son, The i is: 1
...
...
(更多行)
This is son, The i is: 68
This is son, The i is: 69
This is son, The i is: 70
After son, This is father, his pid is: 2623 and the count is: 1, and the child is: 22624
```

从中可以看到，父进程总是等子进程执行完毕后才开始继续执行。

8.2.2 exec 函数

读完 8.2.1 小节，读者可能会产生一个疑惑：既然所有新进程都是由 fork 产生的，而且由 fork 产生的子进程和父进程几乎完全一样，那岂不是意味着系统中所有的进程都应该一模一样了吗？而且，就我们的常识来说，当执行一个新程序的时候，新产生的进程的内容应就是程序的内容才对。是我们理解错了吗？显然不是，要解决这些疑惑，就必须提到下面将要介绍的 exec 函数族。Linux 使用 exec 函数族来执行新的程序，以新的子进程来完全代替原有的进程。

实际上，在 Linux 中并不存在一个 exec()的函数形式，exec 是一个函数族，指的是一组函数，一共有 6 个，分别是：

```
#include <unistd.h>
int execl (const char *pathname, const char *arg, ...);
int execlp (const char *filename, const char *arg, ...);
int execle (const char *pathname, const char *arg, ..., char *const envp[ ]);
int execv (const char *pathname, char *const argv[ ]);
int execvp (const char *filename, char *const argv[ ]);
int execve (const char *pathname, char *const argv[ ], char *const envp[ ]);
```

6 个函数返回：若成功则无返回值，若出错则返回-1。

函数名中含有字母“l”的，其参数个数不定，参数由所调用程序的命令行参数列表组成，最后一个 NULL 表示结束。

函数名中含有字母“v”的，则是使用一个字符串数组指针 argv 指向参数列表，这一字符串数组和含有字母“l”的函数中的参数列表完全相同，也同样以 NULL 结束。

函数名中含有字母“p”的函数可以自动在环境变量 PATH 指定的路径中搜索要执行的程序，因此它的第一个参数为 filename，表示可执行函数的文件名。而其他函数则需要用户在参数列表中指定该程序路径，所以其第一个参数为 pathname。

函数名中含有字母“e”的，比其他函数多含有一个参数 envp，这是一个字符串数组指针，

用于指定环境变量。调用这样的函数(execle 和 execve)时，可以由用户自行设定子进程的环境变量，存放在参数 envp 所指向的字符串数组中。这个字符串数组也必须由 NULL 结束。其他函数则是接收当前环境变量。

事实上，其中只有 execve 才是真正意义上的系统调用，其他都是在此基础上经过包装的库函数。exec 函数族的作用是根据指定的文件名找到可执行文件，并用它来取代调用进程的内容，换句话说，就是在调用进程内部执行一个可执行文件。这里的可执行文件既可以是二进制文件，也可以是任何 Linux 下可执行的脚本文件。

与一般情况不同，exec 函数族的函数执行成功后不会返回，因为调用进程的实体，包括代码段，数据段和堆栈等都已经被新的内容取代，只是进程 ID 等一些表面上的信息仍保持原样，颇有些神似“三十六计”中的“金蝉脱壳”。看上去还是旧的躯壳，却已经注入了新的灵魂。只有调用失败了，它们才会返回一个-1，从原程序的调用点接着往下执行。

现在读者应该大致明白了，Linux 下是如何执行新程序的。每当有进程认为自己不能为系统和用户维护做出任何贡献了，它就可以发挥最后一点余热，调用任何一个 exec，让自己以新的面貌重生；或者，更普遍的情况是，如果一个进程想执行另一个程序，它就可以 fork 出一个新进程，然后调用任何一个 exec，这样看起来就好像通过执行应用程序而产生了一个新进程一样。

事实上第二种情况被应用得更普遍，以至于 Linux 专门为其作了优化，我们已经知道，fork 会将调用进程的所有内容原封不动地复制到新产生的子进程中去，这些复制的动作很消耗时间，而如果 fork 完之后我们马上就调用 exec，这些辛辛苦苦复制来的东西又会被立刻抹掉，这看起来非常不划算，于是人们设计了一种写时复制(Copy-On-Write, COW)的技术，使得 fork 结束后并不立刻复制父进程的内容，而是到了真正实用的时候才复制，这样如果下一条语句是 exec，它就不会白白做无用功了，也就提高了效率。

提 示

帮助读者理解 exec：

exec 的 6 条函数看起来似乎很复杂，但实际上无论是作用还是用法都非常相似，只有很微小的差别。在深入学习它们之前，先来了解一下我们习以为常的 main 函数。下面这个 main 函数的形式可能有些出乎读者的意料：

```
int main(int argc, char *argv[ ], char *envp[ ]);
```

它可能与绝大多数教科书上描述的都不一样，但实际上，这才是 main 函数真正完整的形式。

参数 argc 指出了运行该程序时命令行参数的个数，数组 argv 存放了所有的命令行参数，数组 envp 存放了所有的环境变量。环境变量指的是一组值，从用户登录后就一直存在，很多应用程序需要依靠它来确定系统的一些细节，我们最常见的环境变量是 PATH，它指出了应到哪里去搜索应用程序，如/bin；HOME 也是比较常见的环境变量，它指出了我们在系统中的个人目录。环境变量一般以字符串“XXX=xxx”的形式存在，XXX 表示变量名，xxx 表示变量的值。

值得一提的是，argv 数组和 envp 数组存放的都是指向字符串的指针，这两个数组都以一个 NULL 元素表示数组的结尾。

下面，通过一个程序实例来察看传到 argc、argv 和 envp 里的都是什么东西：

```
/* main.c */
int main(int argc, char *argv[ ], char *envp[ ])
{
printf("### ARGC ###\n%d\n", argc);
    printf("### ARGV ###\n");
    while(*argv)
        printf("%s\n", *(argv++));
    printf("### ENVP ###\n");
    while(*envp)
        printf("%s\n", *(envp++));
     return 0;
}
```

编译它：

```
#gcc -o main main.c
```

运行时，我们故意加几个没有任何作用的命令行参数：

```
#./main -xx 000
### ARGC ###
3
### ARGV ###
./main
-xx
000
### ENVP ###
KDE_MULTIHEAD=false
SSH_AGENT_PID=1888
HOSTNAME=localhost.localdomain
TERM=xterm
SHELL=/bin/bash
HISTSIZE=1000
GTK_RC_FILES=/etc/gtk/gtkrc:/root/.gtkrc:/root/.gtkrc-kde
GS_LIB=/root/.kde/share/fonts
OLDPWD=/home/fighter
QTDIR=/usr/lib/qt-3.1
USER=root
SSH_AUTH_SOCK=/tmp/ssh-XXb30vDj/agent.1832
SESSION_MANAGER=local/localhost.localdomain:/tmp/.ICE-unix/1973
USERNAME=root
KONSOLE_DCOP=DCOPRef(konsole-1999,konsole)
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/root/bin
MAIL=/var/spool/mail/root
KONSOLE_DCOP_SESSION=DCOPRef(konsole-1999,session-1)
PWD=/home/zhangfan/CODE
INPUTRC=/etc/inputrc
XMODIFIERS=@im=Chinput
LANG=zh_CN.GB18030
```

```
GDMSESSION=Default
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
SHLVL=3
HOME=/root
LANGUAGE=zh_CN.GB18030:zh_CN.GB2312:zh_CN
BASH_ENV=/root/.bashrc
LOGNAME=root
LESSOPEN=|/usr/bin/lesspipe.sh %s
DISPLAY=:0.0
G_BROKEN_FILENAMES=1
COLORTERM=
XAUTHORITY=/root/.Xauthority
_=./main
```

当然，对于不同的主机，环境变量数组 envp 中的内容会有所不同。

我们看到，程序将“./main”作为第一个命令行参数，“-xx”作为第二个命令行参数，“000”作为第三个命令行参数，一共有 3 个命令行参数。

现在回过头来看一下 exec 函数族，这里以 execve 为例进行讲解：

```
int execve(const char *path, char *const argv[ ], char *const envp[ ]);
```

对比一下 main 函数的完整形式，可以看出，这两个函数里的 argv 和 envp 是完全一一对应的关系。execve 的第 1 个参数 path 是被执行应用程序的完整路径，第 2 个参数 argv 就是传给被执行应用程序的命令行参数，第 3 个参数 envp 是传给被执行应用程序的环境变量。

留心看一下 exec 的 6 个函数还可以发现，前 3 个函数都是以 execl 开头的，后 3 个都是以 execv 开头的，它们的区别在于：execv 开头的函数是以“char *argv[ ]”这样的形式传递命令行参数，而 execl 开头的函数采用了我们更容易习惯的方式，把参数一个一个列出来，然后以一个 NULL 表示结束。这里的 NULL 的作用和 argv 数组里的 NULL 作用是一样的。

在 6 个函数中，只有 execle 和 execve 使用了“char *envp[ ] ”传递环境变量，其他的 4 个函数都没有这个参数，这并不意味着它们不传递环境变量，这 4 个函数将把默认的环境变量不做任何修改地传给被执行的应用程序。而 execle 和 execve 会用指定的环境变量去替代默认的那些环境变量。

还有 2 个以 p 结尾的函数 execlp 和 execvp，看起来，它们和 execl 与 execv 的差别很小，事实也是如此，除 execlp 和 execvp 之外的 4 个函数都要求它们的第 1 个参数 path 必须是一个完整的路径，如“/bin/ls”；而 execlp 和 execvp 的第 1 个参数 filename 可以简单到仅仅是一个文件名，如“ls”，这两个函数可以自动到环境变量 PATH 制订的目录里去寻找。

程序 8.5 是关于 exec 函数族的应用的例子。代码清单如 exec_example.c 所示。

【程序 8.5】exec 函数族的使用举例：exec_example.c。

```
#include <unistd.h>
#include <stdio.h>
int main(void)
{
char *envp[ ]={"PATH=/tmp","USER=root","STATUS=testing", NULL};
    char *argv_execv[ ]={"echo", "excuted by execv", NULL};
```

```
    char *argv_execvp[ ]={"echo", "executed by execvp", NULL};
    char *argv_execve[ ]={"env", NULL};
    if(fork()==0)
      {
 if(execl("/bin/echo", "echo", "executed by execl", NULL))
            perror("Err on execl");
      }
    if(fork()==0)
      {
          if(execlp("echo", "echo", "executed by execlp", NULL))
            perror("Err on execlp");
      }
    if(fork()==0)
      {
          if(execle("/usr/bin/env", "env", NULL, envp))
             perror("Err on execle");
      }
    if(fork()==0)
      {
          if(execv("/bin/echo", argv_execv))
            perror("Err on execv");
      }
    if(fork()==0)
      {
          if(execvp("echo", argv_execvp))
            perror("Err on execvp");
      }
    if(fork()==0)
      {
          if(execve("/usr/bin/env", argv_execve, envp))
            perror("Err on execve");
      }
    return 0;
}
```

说 明

在程序 8.5 中使用了函数 perror，perror 是一个很常用的函数，在本书的绝大多数出错信息输出时，都可以使用它来代替。perror()用来将上一个函数发生错误的原因输出到标准错误(stderr)。它的函数原型是:

```
#include<stdio.h>
void perror(const char *s);
```

参数 s 所指的字符串会先打印出，后面再加上错误原因字符串。此错误原因依照全局变量 errno 的值来决定要输出的字符串。

程序 8.5 里调用了两个 Linux 常用的系统命令，echo 和 env。echo 会把后面跟的命令行参数原封不动地打印出来，env 用来列出所有环境变量。

使用 gcc 编译 exec_example.c，并生成可执行文件 exec_example：

```
#gcc -o exec_example exec_example.c
```

运行程序，得到输出结果：

```
#./exec_example
executed by execl
executed by execlp
PATH=/tmp
USER=root
STATUS=testing
excuted by execv
executed by execvp
PATH=/tmp
USER=root
STATUS=testing
```

从中可以看到，程序的运行结果与代码是吻合的，对于输出结果的分析，留给读者自己思考(其实已经在上文中进行详细介绍了)。

另外，在程序 8.5 中，由于各个子进程执行的顺序无法控制，所以有可能出现一个比较混乱的输出——各子进程打印的结果交杂在一起，而不是严格按照程序中列出的次序。也就是说，再次运行程序，输出结果可能是像下面这样的：

```
#./exec_example
executed by execl
PATH=/tmp
USER= root
STATUS=testing
executed by execlp
excuted by execv
executed by execvp
PATH=/tmp
USER=lei
STATUS=testing
```

从中可以看到，execle 的输出结果跑到了 execlp 的前面。

注 意

读者在编程过程中，如果用到了 exec 函数族，一定记得要使用错误判断语句。因为与其他系统调用相比，exec 很容易受伤，被执行文件的位置、权限等很多因素都能导致该调用的失败。最常见的错误是：

- 找不到文件或路径，此时 errno 被设置为 ENOENT。
- 数组 argv 和 envp 忘记使用 NULL 结束，此时 errno 被设置为 EFAULT。
- 没有对要执行文件的运行权限，此时 errno 被设置为 EACCES。

8.2.3 exit 和_exit 函数

读者在前面的实例代码中已多次见过 exit 函数的调用。从 exit 的名字就能看出，这个系统调用是用来终止一个进程的。无论在程序中的什么位置，只要执行到 exit 系统调用，进程就会停止剩下的所有操作，清除包括 PCB 在内的各种数据结构，并终止本进程的运行。exit 在 Linux 函数库中的原型是：

```
#include <stdlib.h>
void exit(int status);
```

同 exit 函数一样，_exit 函数也是用于正常终止一个程序，它的函数原型如下：

```
#include <unistd.h>
void _exit(int status);
```

作为系统调用而言，_exit 和 exit 是一对孪生兄弟，它们究竟相似到什么程度，可以从 Linux 的源码中找到答案：

```
#define_NR_exit_NR_exit    /*摘自文件 include/asm-i386/unistd.h 第 334 行*/
```

“__NR_”是在 Linux 的源码中为每个系统调用加上的前缀，请注意第一个 exit 前有 2 条下划线，第二个 exit 前只有 1 条下划线。

同时，exit 和_exit 系统调用都带有一个整数类型的参数 status，我们可以利用这个参数传递进程结束时的状态，比如说，该进程是正常结束的，还是出现某种意外而结束的，一般来说，0 表示没有意外的正常结束；其他的数值表示出现了错误，进程非正常结束。我们在实际编程时，可以用 wait 系统调用接收子进程的返回值，从而针对不同的情况进行不同的处理。关于 wait 的详细情况，将在下一节中进行介绍。

但是，程序员必须清楚的是，_exit 和 exit 之间是有区别的，这种区别主要体现在它们在函数库中的定义。另外，_exit 立即进入内核，exit 则先执行一些清除处理(包括调用执行各终止处理程序，关闭所有标准 I/O 流等)，然后进入内核。使用不同头文件的原因是：exit 是由 ANSI C 说明的，而_exit 则是由 POSIX.1 说明的。

由于历史原因，exit 函数总是执行一个标准 I/O 库的清除关闭操作：对于所有打开流调用 fclose 函数。exit 和_exit 都带一个整型参数 status，称之为终止状态(exit status)。大多数 UNIX shell 都提供检查一个进程终止状态的方法。如果调用这些函数时不带终止状态，或 main 执行了一个无返回值的 return 语句，或 main 执行隐式返回，则该进程的终止状态是未定义的。这就意味着，下列经典性的 C 语言程序是不完整的：

```
#include <stdio.h>
main ()
{
    printf ("hello, world \n");
}
```

因为 main 函数没有使用 return 语句返回(隐式返回)，它在返回到 C 的起动例程时并没有返回一个值(终止状态)。另外，若使用：

```
return(0);
```

或者：

```
exit(0);
```

则向执行此程序的进程(常常是一个 Shell 进程)返回终止状态 0。另外，main 函数的说明实际上应当是：

```
int main(void);
```

将 main 说明为返回一个整型及用 exit 代替 return，对某些 C 编译程序和 UNIX 程序而言会产生不必要的警告信息，因为这些编译程序并不了解 main 中的 exit 与 return 语句的作用相同。警告信息可能是"control reaches end of nonvoid function(控制到达非 void 函数的结束处)"。避开这种警告信息的一种方法是：在 main 中使用 return 语句而不是 exit。但是这样做的结果是不能用 UNIX 的 grep 公用程序来找出程序中所有的 exit 调用。另外一个解决方法是将 main 说明为返回 void 而不是 int，然后仍旧调用 exit。这也避开了编译程序的警告，但从程序设计角度看却并不正确。在本书中的大多数程序，将 main 表示为返回一个整型，因为这是 ANSI C 和 POSIX.1 所定义的。我们将不理会编译程序不必要的警告。

程序 8.6 是关于 exit 函数的最常见的例子。代码如 exit_example.c 所示。

【程序 8.6】exit 函数的使用举例：exit_example.c。

```
#include <stdlib.h>
int main(void)
{
printf("this process will exit!\n");
    exit(0);
    printf("never be displayed!\n");
}
```

使用 gcc 编译 exit_example.c，并生成可执行文件 exit_example：

```
#gcc -o exit_example exit_example.c
```

运行程序，得到输出结果：

```
#./exit_example
this process will exit!
```

从中可以看到，程序并没有打印后面的"never be displayed!\n"，因为在此之前，在执行到 exit(0)时，进程就已经终止了。

exit 函数与_exit 函数最大的区别在于，exit 函数在调用之前要检查文件的打开情况，把文件缓冲区中的内容写回文件；而_exit 直接使进程停止运行，清除其使用的内存空间，并销毁其在内核中的各种数据结构。

在 Linux 的标准函数库中，有一套被称为"高级 I/O"的函数，我们熟知的 printf、fopen、fread、fwrite 都在此列，它们也被称为"缓冲 I/O(buffered I/O)"(参考第 7 章)，其特征是对应每一个打开的文件，在内存中都有一片缓冲区，每次读文件时，会多读出若干条记录，这样下

次读文件时就可以直接从内存的缓冲区中读取，每次写文件的时候，也仅仅是写入内存中的缓冲区，等满足了一定的条件(达到一定数量，或遇到特定字符，如换行符‘\n’和文件结束符EOF)，再将缓冲区中的内容一次性写入文件，这样就大大增加了文件读写的速度，但也为程序员的编程工作带来了一定的麻烦。如果有一些数据，我们认为已经写入了文件，实际上因为没有满足特定的条件，它们还只是保存在缓冲区内，这时如果用_exit 函数直接将进程关闭，缓冲区中的数据就会丢失，反之，如果想保证数据的完整性，就一定要使用 exit 函数。

程序 8.7 正是关于这个问题的一个例子，代码清单如 exit_differ.c 所示。

【程序 8.7】exit 与_exit 函数的区别：exit_differ.c。

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    pid_t pid;
    if ( (pid = fork() )==-1 )
     {
        printf ("failed to create a new process\n");
        exit(0);
     }
    else if(pid==0)    /*子进程*/
     {
        printf("child process, output begin\n");
        printf("child process, content in buffer");     /*没有换行符‘\n’*/
        exit(0);
     }
    else    /*父进程*/
     {
        printf("\nparent process, output begin\n");
        printf("parent process, content in buffer");   /*没有换行符‘\n’*/
        _exit(0);
     }
       return 0;
}
```

使用 gcc 编译 exit_differ.c，并生成可执行文件 exit_differ：

```
#gcc -o exit_differ exit_differ.c
```

运行程序，得到输出结果：

```
#./exit_differ
child process, output begin
child process, content in buffer
parent process, output begin
```

由于 printf 函数在遇到换行符‘\n’时才从缓冲区中读取数据，在子进程中，“exit(0);”在使进程退出之前，先将缓冲区中的内容写到标准输出；而在父进程中，并没有执行

“printf("parent process, content in buffer");”语句，因为“_exit(0);”直接将缓冲区里的内容清洗了。

8.2.4 wait 和 waitpid 函数

在多进程处理时，用户可能需要用到有关进程等待的操作，这种等待可以是进程组成员间的等待，也可以是父进程对子进程的等待。

通过前面的介绍，读者已经了解了父进程和子进程的概念，并已经掌握了系统调用 exit 的用法，但可能很少有人意识到，在一个进程调用了 exit 之后，该进程并非马上就消失掉，而是留下一个称为僵尸进程(Zombie)的数据结构。这时的处理方法之一就是使用进程等待的系统调用 wait 和 waitpid。

说 明

在 Linux 进程的 5 种状态中，僵尸进程是非常特殊的一种，它已经放弃了几乎所有内存空间，没有任何可执行代码，也不能被调度，仅仅在进程列表中保留一个位置，记载该进程的退出状态等信息供其他进程收集，除此之外，僵尸进程不再占有任何内存空间。

wait 和 waitpid 的函数原型是：

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait (int *status);
pid_t waitpid (pid_t pid, int *status, int options);
```

两个函数的返回：若成功则返回进程 ID，若出错则返回-1。

进程一旦调用了 wait，就立即阻塞自己，由 wait 自动分析是否当前进程的某个子进程已经退出，如果让它找到了这样一个已经变成僵尸的子进程，wait 就会收集这个子进程的信息，并把它彻底销毁后返回；如果没有找到这样一个子进程，wait 就会一直阻塞在这里，直到有一个出现为止。

提 示

关于进程的不同状态之间的转换，读者有必要先阅读 Linux 操作系统原理相关的书籍和资料。

参数 status 用来保存被收集进程退出时的一些状态，它是一个指向 int 类型的指针。但如果程序员对这个子进程是如何死掉的毫不在意，而只想把这个僵尸进程消灭掉(事实上绝大多数情况下，程序员们都会这样想)，这时就可以设定这个参数为 NULL，就像下面这样：

```
pid=wait(NULL);
```

如果成功，wait 会返回被收集的子进程的进程 ID，如果调用进程没有子进程，调用就会失败，此时 wait 返回-1，同时 errno 被置为 ECHILD。

下面还是通过一个例子来体验一下 wait 函数的调用。代码如 wait_example.c 所示。

【程序 8.8】wait 函数的使用举例：wait_example.c。

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
int main(void)
{
    pid_t pc,pr;
    if ((pc = fork()) < 0)
     {
        printf("error in fork!");
        exit(1);     /* fork 出错退出*/
     }
    else if(pc==0)   /*子进程*/
{
        printf("This is child process with pid of %d\n",getpid());
        sleep(10);          /*睡眠 10 秒钟*/
     }
    else    /*父进程*/
{
         pr=wait(NULL);       /*在这里等待*/
         printf("I catched a child process with pid of %d\n",pr);
     }
    exit(0);
}
```

使用 gcc 编译 wait_example.c，并生成可执行文件 wait_example：

```
#gcc –o wait_example wait_example.c
```

运行程序，得到输出结果：

```
#./wait_example
This is child process with pid of 2035
I catched a child process with pid of 2035
```

从中可以很明显地看到，在第 2 行结果打印出来前有 10 秒钟的等待时间，这就是我们设定的让子进程睡眠的时间，只有子进程从睡眠中苏醒过来，它才能正常退出，也就才能被父进程捕捉到。其实这里我们不管设定子进程睡眠的时间有多长，父进程都会一直等待下去，读者如果有兴趣的话，可以试着自己修改一下这个数值，看看会出现怎样的结果。

从本质上讲，waitpid 和 wait 的作用是完全相同的，但 waitpid 多出了两个可由用户控制的参数 pid 和 options，从而为用户编程提供了一种更为灵活的方式。waitpid 可以用来等待指定的进程，可以使进程不挂起而立刻返回。参数 pid 用于指定所等待的进程，其取值及相应的含义如表 8.1 所示。

表 8.1　参数 pid 取值及其含义

pid 取值	含　义
pid > 0	只等待进程 ID 为 pid 的子进程，不管其他已经有多少子进程运行结束退出了，只要指定的子进程还没有结束，waitpid 就一直等待下去
pid = -1	等待任何一个子进程退出，没有任何限制，此时 waitpid 等价于 wait
pid = 0	等待同一个进程组中的任何子进程，如果某一子进程已经加入了别的进程组，waitpid 则不会对它做任何理睬
pid < -1	等待一个指定进程组中的任何子进程，这个进程组的 ID 等于 pid 的绝对值

参数 options 提供了一些额外的选项来控制 waitpid，目前在 Linux 中只支持 WNOHANG 和 WUNTRACED 两个选项，这是两个常数，可以用“|”运算符把它们连接起来使用，比如：

```
ret=waitpid (-1,NULL,WNOHANG | WUNTRACED);
```

如果不想使用它们，也可以把 options 设为 0，如：

```
ret=waitpid (-1,NULL,0);
```

如果使用了 WNOHANG 参数调用 waitpid，即使没有子进程退出，它也会立即返回，不会像 wait 那样永远等下去。而 WUNTRACED 参数，由于涉及一些跟踪调试方面的知识，加之极少用到，这里就不进行过多的介绍，有兴趣的读者可以自行查阅相关材料。

看到这里，聪明的读者可能已经看出了端倪——wait 不就是经过包装的 waitpid 吗？没错，察看内核源码目录>/include/unistd.h 文件 349～352 行就会发现以下程序段：

```
static inline pid_t wait(int * wait_stat)
{
    return waitpid(-1,wait_stat,0);
}
```

另外，waitpid 的返回值也要比 wait 稍微复杂一些，一共有 3 种情况：

- 当正常返回的时候，waitpid 返回收集到的子进程的进程 ID。
- 如果设置了选项 WNOHANG，而调用中 waitpid 发现没有已退出的子进程可收集，则返回 0。
- 如果调用中出错，则返回-1，这时 errno 会被设置成相应的值以指示错误所在。

当 pid 所指示的子进程不存在，或此进程存在，但不是调用进程的子进程，waitpid 就会出错返回，这时 errno 被设置为 ECHILD。

程序 8.9 是关于 waitpid 的例子，父进程等待子进程的返回，直到收集到子进程的进程 ID。代码清单如 waitpid_example.c。

【程序 8.9】 waitpid 函数的使用举例：waitpid_example.c。

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
```

```
int main(void)
{
pid_t pc, pr;
    if ( (pc = fork() )==-1 )
      {
            printf ("failed to create a new process\n");
            exit(0);
      }
else if(pc==0)    /*如果是子进程*/
{
            sleep(10);                /*睡眠 10 秒钟*/
            exit(0);
      }
          /*如果是父进程*/
    do{
            pr=waitpid(pc, NULL, WNOHANG);
/*使用了 WNOHANG 参数，waitpid 不会在这里等待*/
            if(pr==0)
{       /*如果没有收集到子进程 */
                  printf("No child exited\n");
                  sleep(1);
             }
         }while(pr==0);       /*没有收集到子进程，就继续尝试*/
      if(pr==pc)
        {
             printf("successfully get child %d\n", pr);
        }
      else
        {
             printf("some error occured\n");
        }
      return 0;
}
```

使用 gcc 编译 waitpid_example.c，并生成可执行文件 waitpid_example：

```
#gcc –o waitpid_example waitpid_example.c
```

运行程序，得到输出结果：

```
#./waitpid_example
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
successfully get child 2078
```

从中可以看到，父进程经过 10 次失败的尝试之后，终于收集到了退出的子进程。

说 明

因为这只是一个例子程序，不便写得太复杂，所以我们就让父进程和子进程分别睡眠了 10 秒钟和 1 秒钟，代表它们分别做了 10 秒钟和 1 秒钟的工作。父、子进程都有工作要做，父进程利用工作的简短间歇察看子进程是否退出，如退出就收集它。

另外，对于参数 status，当其值不为 NULL 时，wait 就会把子进程退出时的状态取出并存入其中，这是一个整数值(int)，指出了子进程是正常退出还是被非正常结束的(一个进程也可以被其他进程用信号结束，这将在第 9 章中进行介绍)，以及正常结束时的返回值，或被哪一个信号结束的等信息。由于这些信息被存放在一个整数的不同二进制位中，所以用常规的方法读取会非常麻烦，人们就设计了一套专门的宏(macro)来完成这项工作，宏定义及其含义如表 8.2 所示，其中最常用的是 WIFEXITED(status)和 WEXITSTATUS(status)两个宏。

表 8.2 status 的宏定义及其含义

宏 定 义	含 义
WIFEXITED(status)	子进程正常退出时，返回一个非零值
WIFSIGNALED(status)	子进程由于接收到信号而退出时，返回一个非零值。但如果进程接收到信号时调用 exit 函数结束，则返回零值
WIFSTOPPED(status)	在调用 waitpid 时选择了 WUNTRACED 选项，且子进程使用 waitpid 返回，则这个宏的返回为一个非零值
WSTOPSIG(status)	当 WIFSTOPPED 为真(非零值)时，将获得停止该进程的信号
WTERMSIG(status)	当 WIFSIGNALED 为真时，将获得终止该进程的信号
WEXITSTATUS(status)	当 WIFEXITED 为真时，此宏才可以使用，返回该进程退出的代码

注 意

虽然名字一样，但表 8.2 中的参数 status 并不同于 wait 唯一的参数——指向整数的指针 status，而是 status 指针所指向的那个整数，切记不要搞混了。

WIFEXITED(status)这个宏用来指示子进程是否为正常退出，如果是，它会返回一个非零值，否则返回零值。

当 WIFEXITED 返回非零值时，可以使用 WEXITSTATUS(status)这个宏来提取子进程的返回值，比如如果子进程调用 exit(5)退出，WEXITSTATUS(status)就会返回 5；如果子进程调用 exit(7)，WEXITSTATUS(status)就会返回 7。有一点值得注意的是，如果进程不是正常退出的，也就是说，WIFEXITED 返回 0，这个值就毫无意义。

程序 8.10 演示了最常用了两个宏的使用，代码清单如 get_status.c。

【程序 8.10】获取子进程的返回状态：get_status.c。

```
#include <sys/types.h>
#include <sys/wait.h>
```

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
	int status;
	pid_t pc,pr;
	if ((pc = fork()) < 0)
	  {
		printf("error in fork!");
		exit(1);	/*fork 出错退出*/
	  }
	else if(pc==0)
{  /*子进程*/
		printf("This is child process with pid of %d.\n",getpid());
		exit(3);  /*子进程返回 3 */
	  }
	else
{	/*父进程*/
		pr=wait(&status);
		if(WIFEXITED(status))
{	/*如果 WIFEXITED 返回非零值*/
				printf("the child process %d exit normally.\n",pr);
				printf("the return code is %d.\n",WEXITSTATUS(status));
			}
else
{	/*如果 WIFEXITED 返回零 */
				printf("the child process %d exit abnormally.\n",pr);
}
	  }
return 0;
}
```

使用 gcc 编译 get_ status.c，并生成可执行文件 get_ status：

```
#gcc –o get_ status get_ status.c
```

运行程序，得到输出结果：

```
#./get_ status
This is child process with pid of 2506.
the child process 2506 exit normally.
the return code is 3.
```

从中可以看到，父进程准确捕捉到了子进程的返回值 3，并把它打印了出来。

说 明

WIFEXITED(status)和 WEXITSTATUS(status)是使用较多的两个宏，其他的宏定义在平时的编程中很少用到，这里也就不再一一详细介绍了，有兴趣的读者可以自己参阅 Linux man pages 去了解它们的用法。

此外，Linux 系统还提供了另外两个用于进程等待的调用——wait3 和 wait4 系统调用。wait3 和 wait4 函数分别相当于 wait 和 waitpid 函数，但和 wait 和 waitpid 相比，wait3 和 wait4 多一个结构指针参数 rusage。它们的函数原型如下：

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>
pid_t wait3 ( int *status, int options, struct rusage *rusage);
pid_t wait4 ( pid_t pid, int *status, int options, struct rusage *rusage);
```

两个函数返回：若成功则返回进程 ID，若出错则返回-1。

wait3 和 wait4 在以前的 waitpid 等函数的基础上增加了可以获取进程及其子进程所占用的 resources 的情况的功能。参数 rusage 是一个结构指针，调用这两个函数时，如果 rusage 不为 NULL，则关于子进程执行时的相关信息将被写入该指针指向的缓冲区内。

鉴于 wait3 和 wait4 的功能与上面讲到的 wait 和 waitpid 的功能十分相似，这里就不再赘述。

8.2.5 进程的一生

下面用一个形象的比喻来对进程短暂的一生做一个小小的总结。

首先，随着 fork 的成功执行，一个新的子进程诞生，但此时的它还只是父进程的一个克隆，从父进程那里得到数据段和堆栈段的复制。然后随着 exec，新进程脱胎换骨，离家独立，开始独自执行一个全新的程序，并完全替代了原有的父进程。

人有生老病死，进程也一样，它可以是自然死亡，即运行到 main 函数的最后一个“}”，从容地离我们而去；也可以是自杀，自杀有两种方式，一种是调用 exit 函数，一种是在 main 函数内使用 return，无论哪一种方式，它都可以留下遗书，放在返回值里保留下来；它甚至还可能被谋杀，被其他进程通过另外一些方式结束它的生命。

进程死掉以后，会留下一具僵尸，wait 和 waitpid 充当了殓尸工，把僵尸推去火化，使其最终归于无形。

这就是进程完整的一生。

8.2.6 用户 ID 和组 ID

在 Linux 系统中，进程的安全性是一个十分重要的问题，同时也是容易使用户糊涂的问题。一个进程的安全性，大体来说，包括该进程的用户 ID 和其用户组 ID 等内容(至少在大部分情况下，用户只需要知道进程的这两个属性)。

与某一个进程相关联的 ID 至少有 6 个，它们分别是实际用户 ID、实际组 ID、有效用户 ID、有效组 ID、保存的设置用户 ID 和保存的设置组 ID。并且，在通常情况下，有效用户 ID 等于实际用户 ID，有效组 ID 等于实际组 ID。

先来看一些获得进程相关信息的函数：

```
#include <sys/types.h>
#include <unistd.h>
uid_t getuid(void);
```

```
gid_t getgid(void);
uid_t gettuid(void);
gid_t getegid(void);
```

4 个函数的返回：进程的相关用户 ID。

其中，getuid 用于获得实际用户标识符；getgid 用于获得实际组标识符；gettuid 用于获得有效用户标识符；getegid 用于获得有效组标识符。

在 Linux 系统中，特权(例如能改变某一文件的读写权限)是基于用户和组 ID 的。当程序需要增加特权，或访问当前并不允许访问的资源时，我们需要更换自己的用户 ID 或组 ID，使得新的 ID 具有适合的特权或访问权限。与此类似，当程序需要降低其特权或阻止对某些资源的访问时，也需要更换用户 ID 或组 ID，从而使得新 ID 不具有相应特权或访问这些资源的能力。下面介绍这些设置或更改用户 ID 和组 ID 的函数调用。

可以使用 setuid 函数设置实际用户 ID 和有效用户 ID。与此类似，可以用 setgid 函数设置实际组 ID 和有效组 ID。它们的函数原型如下：

```
#include <sys/types.h>
#include <unistd.h>
int setuid (uid_t uid);
int setgid (gid_t gid);
```

两个函数返回：若成功则返回 0，若出错则返回-1。

关于谁能更改 ID 有若干规则。现在先考虑有关改变用户 ID 的规则(在这里关于用户 ID 所说明的一切都适用于组 ID)。

- 若进程具有超级用户特权，则 setuid 函数将实际用户 ID、有效用户 ID，以及保存的设置用户 ID 设置为 uid(对于 setgid 则为 gid)。
- 若进程没有超级用户特权，但是 uid 等于实际用户 ID 或保存的设置用户 ID，则 setuid 只将有效用户 ID 设置为 uid(对于 setgid 则为 gid)。不改变实际用户 ID 和保存的设置用户 ID。

如果上面两个条件都不满足，则 errno 设置为 EPERM，并返回出错。

关于内核所维护的 3 个用户 ID，还要注意下列几点：

- 只有超级用户进程可以更改实际用户 ID。通常，实际用户 ID 是在用户登录时，由 login(1)程序设置的，而且绝不会改变它。因为 login 是一个超级用户进程，当它调用 setuid 时，设置所有 3 个用户 ID。
- 仅当对程序文件设置了设置用户 ID 位时，exec 函数设置有效用户 ID。如果设置用户 ID 位没有设置，则 exec 函数不会改变有效用户 ID，而将其维持为原先值。任何时候都可以调用 setuid，将有效用户 ID 设置为实际用户 ID 或保存的设置用户 ID。但不能将有效用户 ID 设置为任一随机值。
- 保存的设置用户 ID 是由 exec 从有效用户 ID 复制的。在 exec 按文件用户 ID 设置了有效用户 ID 后，即进行这种复制，并将此副本保存起来。

此外，setreuid 和 setregid 的功能是交换用户 ID。具体来说，setreuid 用于交换实际用户 ID 和有效用户 ID 的值，setregid 用于交换实际组 ID 和有效组 ID 的值。函数原型如下：

```
#include <sys/types.h>
#include <unistd.h>
int setreuid (uid_t ruid, uid_t euid);
int setregid (gid_t rgid, gid_t egid);
```

两个函数返回：若成功则返回 0，若出错则返回-1。

参数 ruid 和 rgid 表示实际用户 ID 和实际组 ID，euid 和 egid 表示有效用户 ID 和有效组 ID。setreuid 和 setregid 的作用很简单：一个非特权用户总能交换实际用户(或组)ID 和有效用户(或组)ID。这就允许一个设置用户 ID 程序转换成只具有用户的普通许可权，以后又可再次转换回设置用户 ID 所得到的额外许可权。

另外，seteuid 和 setegid 函数用于更改有效用户 ID 和有效组 ID。函数原型如下：

```
#include <sys/types.h>
#include <unistd.h>
int seteuid (uid_t euid);
int setegid (gid_t egid);
```

两个函数返回：若成功则返回 0，若出错则返回-1。

一个非特权用户可将其有效用户 ID 设置为其实际用户 ID 或其保存的设置用户 ID。对于一个特权用户则可将有效用户 ID 设置为 uid(这一点区别于 setuid 函数，它更改 3 个用户 ID)。

8.2.7 system 函数

system 函数是一个和操作系统紧密相关的函数，用户可以使用它在自己的程序中调用系统提供的各种命令。

执行系统的命令行，其实也是调用程序创建一个进程来实现的。实际上，system 函数的实现正是通过调用 fork、exec 和 waitpid 函数来完成的。system 的函数原型如下：

```
#include <stdlib.h>
int system (const char *cmdstring);
```

由于 system 在其实现中调用了 fork、exec 和 waitpid，因此有 3 种可能的返回值：

- 如果 fork 失败或者 waitpid 返回除 EINTR 之外的出错，则 system 返回-1，而且 errno 中设置了错误类型。
- 如果 exec 失败(表示不能执行 Shell)，则其返回值如同 Shell 执行了 exit (127)一样，即返回值为 127。
- 否则所有 3 个函数(fork、exec 和 waitpid)都成功，并且 system 的返回值是 shell 的终止状态，其格式已在 8.2.4 小节中说明。

system 函数的使用是很方便的。system 调用 fork 产生子进程，由子进程来调用/bin/sh-c cmdstring 来执行参数 cmdstring 字符串所代表的命令，此命令执行完后随即返回原调用的进程。在调用 system 期间 SIGCHLD 信号(信号将在第 9 章介绍)会被暂时搁置，SIGINT 和 SIGQUIT 信号则会被忽略。

注 意

如果参数 cmdstring 是一个空指针 NULL，则仅当命令处理程序可用时，system 返回非 0 值，这一特征可以决定在一个给定的操作系统上是否支持 system 函数(也就是说，当 system 函数返回值为 0 时，表明 system 无效)。在类 UNIX 的系统中(比如 Linux)，system 总是可用的。

另外，值得注意的一点是，在编写具有 suid/sgid 权限的程序时请勿使用 system，system 会继承环境变量，通过环境变量可能会造成系统安全的问题。

程序 8.11 是一个关于 system 函数调用的例子。程序中 4 次调用 system，代表命令行字符串的参数 cmdstring 分别被设为不同的值，则 system 返回不同的结果。代码清单如 cmd_system.c 所示。

【程序 8.11】system 函数执行系统命令：cmd_system.c。

```
#include <stdlib.h>
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
      int status;
      if ((status=system(NULL)) < 0)
        {
          printf("system error!\n");
          exit(0);
        }
      printf("exit status=%d\n",status);
      if ((status=system("date")) < 0)
        {
          printf("system error!\n");
          exit(0);
        }
      printf("exit status=%d\n",status);
      if ((status=system("invalidcommand")) < 0)
        {
          printf("system error!\n");
          exit(0);
        }
      printf("exit status=%d\n",status);
      if ((status=system("who;exit 44")) < 0)
        {
          printf("system error!\n");
          exit(0);
        }
      printf("exit status=%d\n",status);
      return 0;
}
```

使用 gcc 编译 cmd_system.c，并生成可执行文件 cmd_system：

```
#gcc -o cmd_system cmd_system.c
```

运行程序，得到输出结果：

```
#./ cmd_system
exit status=1
四 10 月  8 17:49:35 CST 2009
exit status=0
sh: line 1: invalidcommand: command not found
exit status=32512
root        :0           Oct   8 17:06
root        pts/0        Oct   8 17:07
root        pts/1        Oct   8 17:08
exit status=11264
```

如运行结果所示，第 1 次调用 system 时，参数 cmdstring 为空指针 NULL，返回结果 status 为 1(非 NULL 指针)，说明本 Linux 系统下，system 是可用的；第 2 次调用 system，参数 cmdstring 为 Linux 系统下的命令“date”，system 成功执行；第 3 次调用，参数 cmdstring 为一个非法的字符串命令，system 的返回值为 shell 的终止状态(命令出错)32512，即“command not found”；第四次调用，使用“who”命令显示登录到系统的用户情况，“exit 44”是退出当前的 shell，可以看到，system 成功返回，返回值为 11264。

8.3 多个进程间的关系

在一个 Linux 系统中，通常总是有多个进程在同时运行，如何协调各个进程之间的关系，让它们井然有序地执行是评价操作系统优良性的一个很重要的因素。本节将向读者介绍 Linux 内核中几个比较重要的概念，包括进程组、会话期和控制终端等。

8.3.1 进程组

每个进程除了有一个进程 ID 之外，还隶属于一个进程组。进程组是一个或多个进程的集合。每个进程组有一个唯一的进程组 ID。进程组 ID 类似于进程 ID——它是一个正整数，并可存放在 pid_t 的数据类型中。函数 getpgrp 返回调用进程的进程组 ID：

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpgrp (void);
```

返回：调用进程的进程组 ID。

每个进程组都有一个组长进程 leader。组长进程 leader 的标识是其进程组 ID 等于其进程 ID。

进程组组长可以创建一个进程组，也可以创建该组中的进程，然后终止。只要在某个进程组中有一个进程存在，则该进程组就存在，这与其组长进程是否终止无关。从进程组创建开始

到其中最后一个进程离开为止的时间区间称为进程组的生命期。某个进程组中的最后一个进程可以终止，也可以参加另一个进程组。

进程调用 setpgid 可以参加一个现存的组或者创建一个新进程组(下一节中将说明用 setsid 也可以创建一个新的进程组)。

```
#include <sys/types.h>
#include <unistd.h>
int setpgid (pid_t pid, pid_t pgid);
```

返回：若成功则返回 0，出错返回-1。

函数将 pid 进程的进程组 ID 设置为 pgid。如果这两个参数相等，则由 pid 指定的进程变成进程组组长。

一个进程只能为它自己或它的子进程设置进程组 ID。在它的子进程中调用了 exec 之后，它就不能再改变该子进程的进程组 ID 了。

如果 pid 是 0，则使用调用者的进程 ID。另外，如果 pgid 是 0，则由 pid 指定的进程 ID 被当作进程组 ID。

在大多数作业控制 Shell 中，在 fork 之后调用此函数，使父进程设置其子进程的进程组 ID，然后使子进程设置其自己的进程组 ID。这些调用中有一个是冗余的，但这样做可以保证父、子进程在进一步操作之前，子进程都进入了该进程组。如果不这样做的话，那么就产生一个静态条件，因为它依赖于那一个进程先执行。

在讨论信号时，将说明如何将一个信号送给一个进程(由其进程 ID 标识)或送给一个进程组(由进程组 ID 标识)。同样，waitpid 则可被用来等待一个进程或者指定进程组中的一个进程。

8.3.2　会话期

会话期(session)是一个或多个进程组的集合。一个会话可包含多个进程组，但只能有一个前台进程组。例如，可以有如图 8.1 所示的安排。在一个会话期中有 3 个进程组，其中进程 proc1 和 proc2 属于同一个后台进程组，进程 proc3、proc4、proc5 属于同一个前台进程组，shell 进程本身属于一个单独的进程组。

这些进程组的控制终端相同，它们属于同一个 session。当用户在控制终端输入特殊的控制键(例如 Ctrl-C)时，内核会发送相应的信号(例如 SIGINT)给前台进程组的所有进程。各进程、进程组、session 的关系如图 8.1 所示。

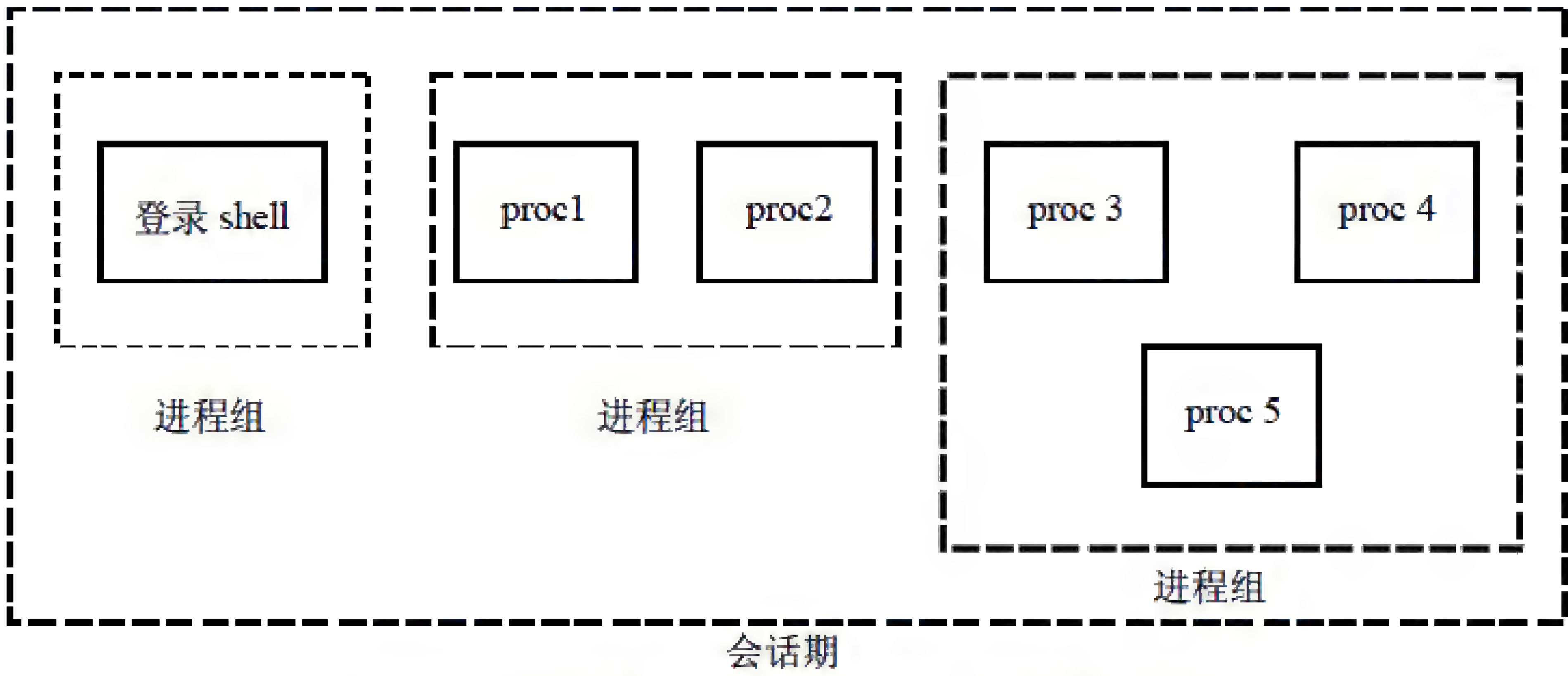


图 8.1　进程组中和会话期中的进程安排

这种安排通常是由 shell 的管道线将几个进程编成一组的。例如，图 8.1 中的安排可能是由下列形式的 shell 命令形成的：

```
procl | proc2 &
proc3 | proc4 | proc5
```

进程调用 setsid 函数就可建立一个新会话期，setsid 的函数原型如下：

```
#include <sys/types.h>
#include <unistd.h>
pid_t setsid (void);
```

返回：若成功则返回进程组 ID，若出错则返回-1。

如果调用此函数的进程不是一个进程组的组长进程 leader，则此函数创建一个新的会话期，结果为：

- 此进程变成该新会话期的会话期首进程(session leader，会话期首进程是创建该会话期的进程)。此进程是该新会话期中的唯一进程。
- 此进程成为一个新进程组的组长进程。新进程组 ID 是此调用进程的进程 ID。
- 此进程没有控制终端(下一小节讨论控制终端)。如果在调用 setsid 之前此进程有一个控制终端，那么这种联系也被解除。

如果此调用进程已经是一个进程组的组长，则此函数返回出错。为了保证不处于这种情况，通常先调用 fork，然后使其父进程终止，而子进程则继续。因为子进程继承了父进程的进程组 ID，而其进程 ID 则是新分配的，两者不可能相等，所以这就保证了子进程不是一个进程组的组长。

下面从会话期和进程组的角度来分析系统登录和执行命令的过程。

getty 或 telnetd 进程在打开终端设备之前调用 setsid 函数创建一个新的会话，该进程称为会话期首进程(Session Leader)，该进程的 ID 也可以看作是 session 的 ID，然后该进程打开终端设备作为这个会话中所有进程的控制终端。在创建新会话的同时也创建了一个新的进程组，该进程是这个进程组的组长进程(Process Group Leader)，该进程的 ID 也是进程组的 ID。

在登录过程中，getty 或 telnetd 进程变成 login，然后变成 shell，但仍然是同一个进程，仍然是 Session Leader。

由 Shell 进程 fork 出的子进程本来具有和 Shell 相同的会话期、进程组和控制终端，但是 Shell 调用 setpgid 函数将作业中的某个子进程指定为一个新进程组的 Leader，然后调用 setpgid 将该作业中的其他子进程也转移到这个进程组中。如果这个进程组需要在前台运行，就调用 tcsetpgrp 函数(随后将向读者介绍)将它设置为前台进程组，由于一个 session 只能有一个前台进程组，所以 Shell 所在的进程组就自动变成后台进程组。

在图 8.1 所示的例子中，proc3、proc4、proc5 被 shell 放到同一个前台进程组，其中有一个进程是该进程组的 Leader，Shell 调用 wait 等待它们运行结束。一旦它们全部运行结束，shell 就调用 tcsetpgrp 函数将自己提到前台继续接受命令。但是注意，如果 proc3、proc4、proc5 中的某个进程又 fork 出子进程，子进程也属于同一进程组，但是 Shell 并不知道子进程的存在，

也不会调用 wait 等待它结束。换句话说，proc3 | proc4 | proc5 是 Shell 的作业，而这个子进程不是，这是作业和进程组在概念上的区别。一旦作业运行结束，Shell 就把自己提到前台，如果原来的前台进程组还存在(如果这个子进程还没终止)，则它自动变成后台进程组。

8.3.3 控制终端

当登录 Linux 系统时，将自动建立控制终端(controlling terminal)。读写控制终端的方法是打开文件/dev/tty(参考 6.1.2 小节中的设备文件)，在内核中，此特殊文件是控制终端的同义语。通常情况下，对于标准输入、标准输出及标准出错，程序都要与控制终端实现交互。

会话期和进程组有一些其他特性，这些特性如图 8.2 所示。

- 一个会话期可以有一个单独的控制终端。这通常是我们在其上登录的终端设备(终端登录情况)或伪终端设备(网络登录情况)。
- 建立与控制终端连接的会话期首进程，被称之为控制进程(controlling process)。
- 一个会话期中的几个进程组可被分成一个前台进程组(foreground process group)及一个或几个后台进程组(background process group)。
- 如果一个会话期有一个控制终端，则它有一个前台进程组，其他进程组则为后台进程组。
- 无论何时按中断键(常常是 Delete 或 Ctrl-C)或退出键(常常是 Ctrl-\)，就会造成将中断信号或退出信号送至前台进程组的所有进程。
- 如果终端界面检测到调制解调器已经脱开连接，则将挂断信号送至控制进程(会话期首进程)。

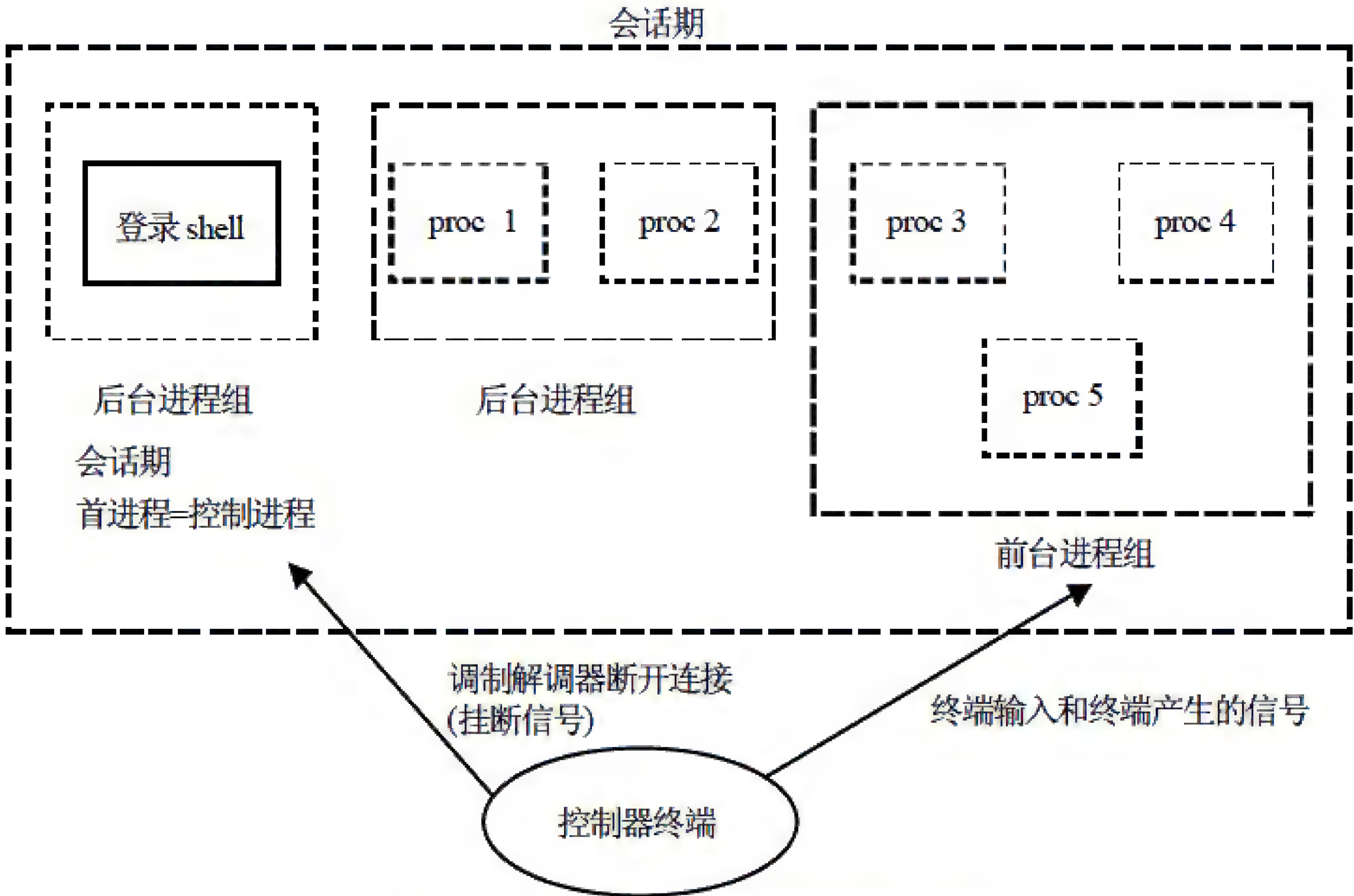


图 8.2　进程组、会话期和控制终端

另外，前面已讲到，在一个会话期中仅有一个前台进程组，那么就需要有一种方法来通知内核哪一个进程组是前台进程组，这样，终端设备驱动程序就能了解将终端输入和终端产生的信号送到何处(见图 8.2)。相关的系统调用为 tcgetpgrp 和 tcsetpgrp。tcgetpgrp 返回前台进程组的

ID，函数原型如下：

```
#include <sys/types.h>
#include <unistd.h>
pid_t tcgetpgrp (int fd);
```

返回：若成功则返回前台进程组 ID，若出错则返回-1。

tcsetpgrp 用于将某一进程组设置为前台进程组，函数原型如下：

```
#include <sys/types.h>
#include <unistd.h>
int tcsetpgrp (int fd, pid_t pgrpid);
```

返回：若成功则返回 0，若出错则返回-1。

函数 tcgetpgrp 返回前台进程组 ID，它与在 fd 上打开的终端相关。

如果进程有一个控制终端，则该进程可以调用 tcsetpgrp 将前台进程组 ID 设置为 pgrpid。pgrpid 值应当是在同一会话期中的一个进程组的 ID。fd 必须引用该会话期的控制终端。大多数应用程序并不直接调用这两个函数，它们通常由作业控制 Shell 调用。

最后，需要提醒读者的是，关于进程的优先级及多个进程间的轮换调度算法，并不属于本书的内容，要掌握这部分的知识，有必要参考操作系统原理方面的书籍。

8.4 本章小结

本章主要讲述了 Linux 进程的概念，以及 Linux 下进程控制的相关函数调用，最后还讲到了多个进程间的关系。进程是具有一定功能的正在运行着的程序，是关于数据集合的一次执行过程，它是现代操作系统中一个十分重要的概念。Linux 系统本身是一个支持多进程的操作系统，了解和掌握 Linux 下进程控制的相关系统调用，以及多个进程之间相互协调运行的关系原理，对于程序员进行 Linux 下的程序开发是有很大益处的。

实战演练

1. 使用 ps 命令查看自己的系统中目前有多少进程正在运行，再使用“ps-aux”命令查看当前系统中每一个进程的详细信息。
2. 在 Linux 下，使用 kill 命令用来中止一个进程，试使用 kill 命令来关闭当前的某个进程。
3. 编写一个程序，获取当前进程的进程 ID。
4. 编写一个程序，使用 fork 函数创建一个子进程，并分别获取父、子进程的进程 ID。
5. 编写一个程序，使用 vfork 函数创建一个子进程，在子进程中执行变量 count 的加 1 运算，当子进程执行完毕后，由父进程返回变量 count 的值。
6. 编写一个程序，考察 exit 函数的使用方法，在程序尚未运行到最后时使用 exit 函数退出，

看后面的程序语句是否会被执行？

7. 编写一个程序，考察_exit 函数的使用方法，在程序运行到最后时使用_exit 函数退出，看程序的执行结果会发生怎样的变化。

8. 编写一个程序，创建一个子进程，使用 wait 函数让父进程等待子进程的结束，并打印出父进程的等待信息。

9. 编写一个程序，使用 system 函数来执行 Linux 的 shell 命令，比如“data”、“ls”、“pwd”等。

10. 编写一个程序，获取当前进程的进程组 ID。

第 9 章

信　　号

通过上一章的介绍，读者已经了解到，Linux 是一个多用户多进程的操作系统，系统中同时存在运行着多个进程。在一个进程的运行过程中，如果某一事件发生，系统需要通知这个正在运行的进程，其中一个很常用的方法就是使用信号。信号是一种进程间通信的机制，其显著的特点是，信号不是用于将数据发送给某一进程，而是用于通知一个进程某一特定事件的发生。一般来说，大多数的应用程序都涉及信号的使用，信号是 Linux 编程中非常重要的部分。本章将详细介绍信号机制的基本概念、Linux 对信号机制的大致实现方法、如何使用信号，以及有关信号的几个系统调用。



本章内容：

◎　Linux 信号的基本概念。
◎　Linux 信号处理的机制。
◎　信号操作的相关系统调用。

9.1 Linux 信号简介

信号机制是进程之间相互传递消息的一种方法，信号全称为软中断信号，也有人称作软中断。从它的命名可以看出，它的实质和使用很像中断。所以，信号可以说是进程控制的一部分。本节主要介绍信号的一些基本概念，然后给出一些基本的信号类型和信号对应的事件，以及信号的处理机制。基本概念是正确理解和使用信号的关键。

9.1.1 信号的基本概念

信号是一种进程间通信的方法，应用于异步事件的处理。信号的实质是一种软中断，它被发送给一个正在被执行的进程以通知该进程有某一事件发生了。本小节向读者介绍信号的基本含义和分类，以及 Linux 下的信号列表。

1. 信号的含义

软中断信号(signal，又简称为信号)用来通知进程发生了异步事件。进程之间可以互相通过系统调用 kill()发送软中断信号。内核也可以因为内部事件而给进程发送信号，通知进程发生了某个事件。注意，信号只是用来通知某进程发生了什么事件，并不给该进程传递任何数据。

收到信号的进程对各种信号有不同的处理方法。处理方法可以分为 3 类：第一种是类似中断的处理程序，对于需要处理的信号，进程可以指定处理函数，由该函数来处理。第二种方法是忽略某个信号，对该信号不进行任何处理，就像从未发生过一样。第三种方法是对该信号的处理保留系统的默认值，这种默认操作大多数是使得进程终止。进程通过系统调用 signal()来指定进程对某个信号的处理行为。

在进程表的表项中有一个软中断信号域，该域中每一位对应一个信号，当有信号发送给进程时，对应位置位。由此可以看出，进程对不同的信号可以同时保留，但对于同一个信号，进程并不知道在处理之前来过多少个。

每个信号都有一个名字，这些名字都以 3 个字符 SIG 开头。例如，SIGABRT 是夭折信号，当进程调用 abort()函数时(会使进程非正常结束)产生这种信号。SIGALRM 是闹钟信号，当由 alarm()函数设置的时间已经超过后产生此信号。在头文件<signal.h>中，这些信号都被定义为正整数，称为信号编号。没有编号为 0 的信号，kill()函数对编号 0 有特殊的应用，POSIX.1 将此种信号编号值称为空信号。

2. 信号分类

发出信号的原因很多，这里按发出信号的原因简单分类，以了解各种信号：

- 与进程终止相关的信号。当进程退出，或者子进程终止时，发出这类信号。
- 与进程例外事件相关的信号。如进程越界，或企图写一个只读的内存区域(如程序正文区)，或执行一个特权指令及其他各种硬件错误。
- 与在系统调用期间遇到不可恢复条件相关的信号。如执行系统调用 exec 时，原有资源已经释放，而目前系统资源又已经耗尽。
- 与执行系统调用时遇到非预测错误条件相关的信号。如执行一个并不存在的系统调用。

➢ 在用户态下的进程发出的信号。如进程调用系统调用 kill 向其他进程发送信号。
➢ 与终端交互相关的信号。如用户关闭一个终端，或按“break”键等情况。
➢ 跟踪进程执行的信号。

3. 信号列表

可以通过在 Shell 下键入“kill -l”命令查看系统的信号列表，或者键入“man 7 signal”查看更详细的说明。Linux-2.4.20 版本内核支持的信号列表如下：

```
# kill -l
 1) SIGHUP        2) SIGINT        3) SIGQUIT       4) SIGILL
 5) SIGTRAP       6) SIGABRT       7) SIGBUS        8) SIGFPE
 9) SIGKILL      10) SIGUSR1      11) SIGSEGV      12) SIGUSR2
13) SIGPIPE      14) SIGALRM      15) SIGTERM      17) SIGCHLD
18) SIGCONT      19) SIGSTOP      20) SIGTSTP      21) SIGTTIN
22) SIGTTOU      23) SIGURG       24) SIGXCPU      25) SIGXFSZ
26) SIGVTALRM    27) SIGPROF      28) SIGWINCH     29) SIGIO
30) SIGPWR       31) SIGSYS       33) SIGRTMIN     34) SIGRTMIN+1
35) SIGRTMIN+2   36) SIGRTMIN+3   37) SIGRTMIN+4   38) SIGRTMIN+5
39) SIGRTMIN+6   40) SIGRTMIN+7   41) SIGRTMIN+8   42) SIGRTMIN+9
43) SIGRTMIN+10  44) SIGRTMIN+11  45) SIGRTMIN+12  46) SIGRTMIN+13
47) SIGRTMIN+14  48) SIGRTMIN+15  49) SIGRTMAX-14  50) SIGRTMAX-13
51) SIGRTMAX-12  52) SIGRTMAX-11  53) SIGRTMAX-10  54) SIGRTMAX-9
55) SIGRTMAX-8   56) SIGRTMAX-7   57) SIGRTMAX-6   58) SIGRTMAX-5
59) SIGRTMAX-4   60) SIGRTMAX-3   61) SIGRTMAX-2   62) SIGRTMAX-1
63) SIGRTMAX
```

列表中，编号为 1～31 的信号为传统 UNIX 支持的信号，是不可靠信号(非实时的)，编号为 32～63 的信号是后来扩充的，被称为可靠信号(实时信号)。不可靠信号和可靠信号的区别在于前者不支持排队，可能会造成信号的丢失，而后者不会。

下面对编号小于 SIGRTMIN 的信号进行简单介绍。

➢ SIGHUP：本信号在用户终端连接(正常或非正常)结束时发出，通常是在终端的控制进程结束时，通知同一会话期(Session)内的各个作业，这时它们与控制终端不再关联(参考 8.3 节)。登录 Linux 时，系统会自动分配给登录用户一个控制终端。在这个终端运行的所有程序，包括前台进程组和后台进程组，一般都属于同一个会话。当用户退出 Linux 登录时，前台进程组和后台有对终端输出的进程将会收到 SIGHUP 信号。这个信号的默认操作为终止进程，因此前台进程组和后台有终端输出的进程就会中止。此外，对于与终端脱离关系的守护进程，这个信号用于通知它重新读取配置文件。
➢ SIGINT：程序终止(或中断，interrupt)信号，在用户键入 INTR 字符(通常是 Ctrl+c 或 Delete 键)时发出，用于通知前台进程组终止进程。
➢ SIGQUIT：和 SIGINT 类似，但由 QUIT 字符(通常是 Ctrl+\)来控制。进程在因收到 SIGQUIT 退出时会产生 core 文件，在这个意义上类似于一个程序错误信号。
➢ SIGILL：执行了非法指令。通常是因为可执行文件本身出现错误，或者试图执行数据段，堆栈溢出时也有可能产生这个信号。

- SIGTRAP：由断点指令或其他陷进(trap)指令产生，由调试器(debugger)使用，比如跟踪陷进信号。
- SIGABRT：调用 abort 函数时产生的信号，将会使进程非正常结束。
- SIGBUS：非法地址，包括内存地址对齐(alignment)出错。比如访问一个 4 个字长的整数，但其地址不是 4 的倍数。它与 SIGSEGV 的区别在于后者是由于对合法存储地址的非法访问触发的(如访问不属于自己存储空间或只读存储空间)。
- SIGFPE：在发生致命的算术运算错误时发出。不仅包括浮点运算错误，还包括溢出及除数为 0 等其他所有的算术的错误。
- SIGKILL：用来立即结束程序的运行。本信号不能被阻塞、处理和忽略。如果管理员发现某个进程终止不了，可尝试发送这个信号。
- SIGUSR1：留给用户使用，可由用户在应用程序中自行定义。
- SIGSEGV：试图访问未分配给登录用户的内存区，或试图向没有写权限的内存地址写数据。
- SIGUSR2：留给用户使用，可由用户在应用程序中自行定义。
- SIGPIPE：管道破裂信号，当对一个读进程已经运行结束的管道执行写操作时产生。这种情况通常发生在进程间通信时，比如采用管道(FIFO)通信(在第 10 章讲解)的两个进程，读管道还没有打开或者意外终止就向管道写时，写进程会收到 SIGPIPE 信号。此外比如使用套接字(Socket)通信(在第 11 章讲解)的两个进程，写进程在写 Socket 的时候，读进程已经终止。
- SIGALRM：时钟定时信号，计算的是实际的时间或时钟时间。由 alarm 函数设定的时间段终止时，会产生该信号。
- SIGTERM：程序结束(terminate)信号，与 SIGKILL 不同的是该信号可以被阻塞和处理。通常用来要求程序自己正常退出，shell 命令“kill”默认产生这个信号。如果进程终止不了，才会尝试 SIGKILL。
- SIGSTKFLT：堆栈错误。
- SIGCHLD：子进程结束时，父进程会收到这个信号。如果父进程没有处理这个信号，也没有等待子进程，子进程虽然终止，但是还会在内核进程表中占有表项，这时的子进程称为僵尸进程(参考 8.2.4 小节)，这种情况我们应该尽量避免。也是就说，父进程或者忽略 SIGCHILD 信号，或者捕捉它，或者等待它派生的子进程，或者父进程先终止，这时子进程的终止自动由 init 进程来接管。
- SIGCONT：让一个停止(stopped)的进程继续执行。此信号不能被阻塞，可以用一个信号处理程序来让程序在由停止状态变为继续执行时完成特定的工作。例如，重新显示提示符。
- SIGSTOP：停止(stopped)进程的执行。注意它和 terminate 及 interrupt 的区别：该进程还未结束，只是暂停执行。此信号不能被阻塞、处理或忽略。

注 意

SIGKILL 和 SIGSTOP 是两个不能被应用程序捕捉和忽略的信号，这是为了使系统管理员能在任何时候结束或停止某一特定进程的执行。

- SIGTSTP：停止进程的运行，但该信号可以被处理和忽略。用户键入 SUSP 字符时(通常是 Ctrl+z)发出这个信号。
- SIGTTIN：当后台作业要从用户终端读数据时，该作业中的所有进程会收到 SIGTTIN 信号，默认时这些进程会停止执行。
- SIGTTOU：类似于 SIGTTIN，但在写终端(或修改终端模式)时收到。
- SIGURG：套接字上出现紧急情况时产生此信号，比如紧急数据。
- SIGXCPU：超过 CPU 时间资源限制时产生的信号。这个限制可以由 getrlimit/setrlimit 来读取/改变。
- SIGXFSZ：当进程企图扩大文件以至于超过文件大小资源限制时产生此信号。
- SIGVTALRM：虚拟时钟信号，类似于 SIGALRM，但是计算的是该进程占用的 CPU 时间。
- SIGPROF：类似于 SIGALRM/SIGVTALRM，但包括该进程使用的 CPU 时间以及系统调用的时间。
- SIGWINCH：窗口大小改变时发出的信号。
- SIGIO：文件描述符准备就绪，表示可以开始进行输入/输出操作。
- SIGPWR：电源失效信号(Power failure)。
- SIGSYS：非法的系统调用。

注 意

- 在以上列出的信号中，程序不可捕获、阻塞或忽略的信号有：SIGKILL、SIGSTOP。
- 不能恢复至默认动作的信号有：SIGILL、SIGTRAP。
- 默认会导致进程流产的信号有：SIGABRT、SIGBUS、SIGFPE、SIGILL、SIGIOT、SIGQUIT、SIGSEGV、SIGTRAP、SIGXCPU、SIGXFSZ。
- 默认会导致进程退出的信号有：SIGALRM、SIGHUP、SIGINT、SIGKILL、SIGPIPE、SIGPOLL、SIGPROF、SIGSYS、SIGTERM、SIGUSR1、SIGUSR2、SIGVTALRM。
- 默认会导致进程停止的信号有：SIGSTOP、SIGTSTP、SIGTTIN、SIGTTOU。
- 默认进程忽略的信号有：SIGCHLD、SIGPWR、SIGURG、SIGWINCH。

此外，SIGIO 在 SVR4 中是退出，在 4.3BSD 中是忽略；SIGCONT 在进程挂起时是继续，否则是忽略，不能被阻塞。信号 SIGIOT 与 SIGABRT 是同一个信号。

另外，同一个信号在不同的系统中值可能不一样，所以建议最好使用为信号定义的名字，而不要直接使用信号的值。

9.1.2　信号处理机制

上面较详细地介绍了信号的基本概念，在这一小节中，将向读者介绍内核如何实现信号机

制，即内核如何向一个进程发送信号、进程如何接收一个信号、进程怎样控制自己对信号的反应、内核在什么时机处理和怎样处理进程收到的信号。还要介绍一下 setjmp 和 longjmp 在信号中起到的作用。

1. 内核对信号的基本处理方法

内核给一个进程发送软中断信号的方法是，在进程所在的进程表项的信号域设置对应于该信号的位(内核通过在进程的 struct task_struct 结构中的信号域中设置相应的位来实现向一个进程发送信号)。这里要补充的是，如果信号发送给一个正在睡眠的进程，那么要看该进程进入睡眠的优先级，如果进程睡眠在可被中断的优先级上，则唤醒进程；否则仅设置进程表中信号域相应的位，而不唤醒进程。这一点比较重要，因为进程检查是否收到信号的时机是：一个进程在即将从内核态返回到用户态时；或者，在一个进程要进入或离开一个适当的低调度优先级睡眠状态时。

内核处理一个进程收到的信号的时机是在一个进程从内核态返回用户态时。所以，当一个进程在内核态下运行时，软中断信号并不立即起作用，要等到将返回用户态时才处理。进程只有处理完信号才会返回用户态，进程在用户态下不会有未处理完的信号。

内核处理一个进程收到的软中断信号是在该进程的上下文中，因此，进程必须处于运行状态。前面介绍概念的时候讲过，处理信号有 3 种类型：进程接收到信号后退出；进程忽略该信号；进程收到信号后执行用户自定义的使用系统调用 signal()注册的函数。当进程接收到一个它忽略的信号时，进程丢弃该信号，就像从来没有收到该信号似的，而继续运行。如果进程收到一个要捕捉的信号，那么进程从内核态返回用户态时执行用户定义的函数。而且执行用户定义的函数的方法很巧妙，内核是在用户栈上创建一个新的层，该层中将返回地址的值设置成用户定义的处理函数的地址，这样进程从内核返回弹出栈顶时就返回到用户定义的函数处，从函数返回再弹出栈顶时，才返回原先进入内核的地方。这样做的原因是用户定义的处理函数不能且不允许在内核态下执行(如果用户定义的函数在内核态下运行的话，用户就可以获得任何权限)。

在信号的处理方法中有几点特别要引起注意。

- 在一些系统中，当一个进程处理完中断信号返回用户态之前，内核清除用户区中设定的对该信号的处理例程的地址，即下一次进程对该信号的处理方法又改为默认值，除非在下一次信号到来之前再次使用 signal()系统调用。这可能会使得进程在调用 signal()之前又得到该信号而导致退出。在 BSD 中，内核不再清除该地址。但不清除该地址可能使得进程因为过多过快的得到某个信号而导致堆栈溢出。为了避免出现上述情况。在 BSD 系统中，内核模拟了对硬件中断的处理方法，即在处理某个中断时，阻止接收新的该类中断。
- 如果要捕捉的信号发生于进程正在一个系统调用中时，并且该进程睡眠在可中断的优先级上，这时该信号引起进程做一次 longjmp()调用(稍后介绍)，跳出睡眠状态，返回用户态并执行信号处理例程。当从信号处理例程返回时，进程就像从系统调用返回一样，但返回了一个错误代码，指出该次系统调用曾经被中断。这里需注意的是，BSD 系统中内核可以自动地重新开始系统调用。

- 若进程睡眠在可中断的优先级上，则当它收到一个要忽略的信号时，该进程被唤醒，但不做 longjmp()调用，一般是继续睡眠。但用户感觉不到进程曾经被唤醒，而是像没有产生过信号一样。
- 内核对子进程终止(SIGCLD)信号的处理方法与其他信号有所区别。当进程检查出收到了一个子进程终止的信号时，默认情况下，该进程就像没有收到该信号似的，如果父进程执行了系统调用 wait，进程将从系统调用 wait 中醒来并返回 wait 调用，执行一系列 wait 调用的后续操作(找出僵死的子进程，释放子进程的进程表项)，然后从 wait 中返回。SIGCLD 信号的作用是唤醒一个睡眠在可被中断优先级上的进程。如果该进程捕捉了这个信号，就像普通信号处理一样转到处理例程。如果进程忽略该信号，那么系统调用 wait 的动作就有所不同，因为 SIGCLD 的作用仅仅是唤醒一个睡眠在可被中断优先级上的进程，那么执行 wait 调用的父进程被唤醒继续执行 wait 调用的后续操作，然后等待其他的子进程。
- 如果一个进程调用 signal 系统调用，并设置了 SIGCLD 的处理方法，并且该进程有子进程处于僵死状态，则内核将向该进程发一个 SIGCLD 信号。

2. setjmp 和 longjmp 的作用

前面在介绍信号处理机制时，多次提到了 setjmp 和 longjmp，但没有仔细说明它们的作用和实现方法。这里就此进行一个简单的介绍。

在介绍信号的时候，我们看到多个地方要求进程在检查收到信号后，从原来的系统调用中直接返回，而不是等到该调用完成。这种进程突然改变其上下文的情况，就是使用 setjmp 和 longjmp 的结果。setjmp 将保存的上下文存入用户区，并继续在旧的上下文中执行。这就是说，进程执行一个系统调用，当因为资源或其他原因要去睡眠时，内核为进程作了一次 setjmp，如果在睡眠中被信号唤醒，进程不能再进入睡眠时，内核为进程调用 longjmp，该操作是内核为进程将原先 setjmp 调用保存在进程用户区的上下文恢复成现在的上下文，这样就使得进程可以恢复等待资源前的状态，而且内核为 setjmp 返回 1，使得进程知道该次系统调用失败。这就是它们的作用。

setjmp 和 longjmp 的函数原型都在<setjmp.h>中。setjmp 的函数原型如下：

```
include <setjmp.h>
int setjmp (jmp_buf envbuf);
```

setjmp 函数用缓冲区 envbuf 保存系统堆栈的内容，以便后续的 longjmp 函数使用。setjmp 函数初次启用时返回 0 值。

longjmp 的函数原型如下：

```
include <setjmp.h>
void longjmp(jmp_buf envbuf, int val);
```

longjmp 函数中的参数 envbuf 是由 setjmp 函数所保存的堆栈环境，参数 val 设置 setjmp 函数的返回值。longjmp 函数本身是没有返回值的，它执行后跳转到保存 envbuf 参数的 setjmp 函数调用，并由 setjmp 函数调用返回，此时 setjmp 函数的返回值就是 val。

调用 longjmp 函数时不能使 setjmp 函数返回 0，如果 val 为 0，则 setjmp 函数返回 1。longjmp

函数从来不返回，因为它调用后就跳转到 setjmp 函数保存的堆栈处，恢复堆栈开始执行，所以 longjmp 函数不会返回。

另外请特别注意，setjmp 函数与 longjmp 函数总是组合起来使用，它们是紧密相关的一对操作，只有将它们结合起来使用，才能达到程序控制流有效转移的目的，才能按照程序员的预先设计的意图，去实现对程序中可能出现的异常进行集中处理。

提 示

setjmp 和 longjmp 是 C 标准库中提供的一对函数，在实际使用的过程中，有很多需要注意的细节，比如两者必须有严格的先后执行顺序(先调用 setjmp 函数，之后再调用 longjmp 函数)；longjmp 的调用是有一定的域范围要求的；setjmp 和 longjmp 并不能很好地支持 C++ 中面向对象的语义等。此类更深入层次的细节，读者可以参考相关的 C/C++书籍和资料。

9.2 信号操作的相关函数

这一节将详细介绍 Linux 下信号处理的相关函数调用，包括注册信号处理函数、信号的发生和信号的阻塞等。

9.2.1 信号的处理

通过前面的讲解，读者已经了解 Linux 内核对信号的处理机制，本小节将介绍信号处理的相关函数调用。

1. signal 函数

要对一个信号进行处理，就需要给出此信号发生时系统所调用的处理函数。可以为一个特定的信号(除去无法捕捉的 SIGKILL 和 SIGSTOP 信号)注册相应的处理函数。如果正在运行的程序源代码里注册了针对某一特定信号的处理程序，不论当时程序执行到何处，一旦进程接收到该信号，相应的调用就会发生。

通过调用 signal 函数来注册某个特定信号的处理程序，它的函数原型如下：

```
#include <signal.h>
void (*signal (int signum, void (*handler) (int) ) ) (int);
```

返回：若成功则返回以前的信号处理配置，若出错则为 SIG_ERR。

说 明

上述声明格式比较复杂，如果不清楚如何使用，也可以通过下面这种类型定义的格式来使用(POSIX 的定义)：

```
typedef void (*sighandler_t) (int);
sighandler_t signal (int signum, sighandler_t handler);
```

但这种格式在不同的系统中有不同的类型定义，所以要使用这种格式，最好还是参考联机

手册。

参数 signum 表示所注册函数针对的信号，其取值为 9.1.1 小节中讲到的信号名。handler 的取值是：常数 SIG_IGN、常数 SIG_DFL 或当接到此信号后要调用的函数的地址。如果指定 SIG_IGN，则向内核表示忽略此信号(注意有两个信号 SIGKILL 和 SIGSTOP 不能忽略)。如果指定 SIG_DFL，则表示接到此信号后的动作是系统默认动作。当指定函数地址时，我们称此为捕捉此信号，并称此函数为信号处理程序(signal handler)或信号捕捉函数(signal-catching function)。

signal 函数的原型说明此函数要求两个参数，返回一个函数指针，而该指针所指向的函数无返回值(void)。第一个参数 signum 是一个整型数，第二个参数是函数指针，它所指向的函数需要一个整型参数，无返回值。用一般语言来描述也就是要向信号处理程序传送一个整型参数，而它却无返回值。当调用 signal 设置信号处理程序时，第二个参数是指向该函数(也就是信号处理程序)的指针。signal 的返回值则是指向以前的信号处理程序的指针。

如果查看系统的头文件<signal.h>，则可能会找到下列形式的说明：

```
#define SIG_ERR ( void ( * ) ( ) ) -1
#define SIG_DFL ( void ( * ) ( ) ) 0
#define SIG_IGN ( void ( * ) ( ) ) 1
```

这些常数可用于表示指向函数的指针，该函数需要一个整型参数，而且无返回值。signal 的第二个参数及其返回值就可用它们表示。这些常数所使用的 3 个值不一定要是-1，0 和 1，但它们必须是 3 个值而决不能是任一可说明函数的地址。大多数 UNIX 系统使用上面所示的值。

下面几个小程序向读者演示了 signal 函数的简单使用。程序 9.1 演示了捕捉终端键入“Ctrl+c”时产生的 SIGINT 信号。

【程序 9.1】捕捉 SIGINT 信号：catch_sigint.c。

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
void SignHandler(int iSignNum)
{
    printf("Capture signal number:%d\n",iSignNum);
    exit(1);
}
int main(void)
{
    signal(SIGINT,SignHandler);
    while(1)
        sleep(1);
    return 0;
}
```

使用 gcc 编译 catch_sigint.c，并生成可执行文件 catch_sigint：

```
#gcc -o catch_sigint catch_sigint.c
```

运行程序，得到输出结果：

```
#./catch_sigint
Capture signal number:2(键入“Ctrl+c”)
Capture signal number:2(键入“Ctrl+c”)
退出(键入“Ctrl+\”)
#                              /*程序正常退出*/
```

从中可以看到，程序运行起来以后，通过按键“Ctrl+c”将不能再终止程序的运行。因为“Ctrl+c”产生的 SIGINT 信号已经由进程中注册的 SignHandler 函数捕捉了。该程序可以通过“Ctrl+\”终止，因为组合键“Ctrl+\”能够产生 SIGQUIT 信号，而该信号的捕捉函数尚未在程序中注册。

程序 9.2 演示了忽略终端键入“Ctrl+c”时产生的 SIGINT 信号。

【程序 9.2】忽略 SIGINT 信号：ignore_sigint.c。

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
int main(void)
{
    signal(SIGINT,SIG_IGN);
    while(1)
        sleep(1);
    return 0;
}
```

该程序运行起来以后，将“Ctrl+c”产生的 SIGINT 信号忽略掉了，所以“Ctrl+c”将不再能使该进程终止，给用户的感觉是“Ctrl+c”不再起任何作用了。要终止该进程，可以向进程发送 SIGQUIT 信号，即组合键“Ctrl+\”。

提 示

读者可以通过以上两个例子体会捕捉信号与忽略信号之间的异同。

程序 9.3 演示了接收信号的默认处理方式，接受默认处理就相当于没有写信号处理程序。

【程序 9.3】SIGINT 信号的默认处理：default_sigint.c。

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
int main(void)
{
    signal(SIGINT, SIG_DFL);
    while(1)
        sleep(1);
    return 0;
}
```

使用 gcc 编译 default_sigint.c，并生成可执行文件 default_sigint：

```
#gcc -o default_sigint default_sigint.c
```

运行程序，得到输出结果：

```
#./default_sigint
(键入“Ctrl+c”，进程终止，输出空行)
```

再次运行程序：

```
#./default_sigint
退出(键入“Ctrl+\”)
```

由此可见，程序运行的结果正是用户所熟悉的系统默认设置。

通常情况下，在一个用户进程中需要处理多个信号，可以在一段程序代码中定义多个信号的处理函数。可以是一个信号对应一个特定的处理函数，还可以是多个信号对应同一个处理函数。

程序 9.4 中注册了 3 个信号，通过不同的信号值来判断终端产生的信号，以执行不同的信号处理函数。源代码如 signals.c 所示。

【程序 9.4】定义多个信号处理函数：signals.c。

```
#include <signal.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

void sigroutine(int dunno)
{
    switch (dunno)
    {
        case 1:printf("Capture SIGHUP signal, the signal number is %d\n", dunno); break;
        case 2:printf("Capture SIGINT signal, the signal number is %d\n", dunno); break;
        case 3:printf("Capture SIGQUIT signal, the signal number is %d\n", dunno);break;
    }
    return;
}

int main(void)
{
    printf("process ID is %d\n",getpid());
    if(signal(SIGHUP, sigroutine) == SIG_ERR)      /*下面设置了 3 个信号的处理方法*/
    {
        printf("Couldn't register signal handler for SIGHUP!\n");
    }
    if(signal(SIGINT, sigroutine)==SIG_ERR)
    {
        printf("Couldn't register signal handler for SIGINT!\n");
    }
    if(signal(SIGQUIT, sigroutine)==SIG_ERR)
    {
        printf("Couldn't register signal handler for SIGQUIT!\n");
```

```
    }
    while(1)
        sleep(1);
    return 0;
}
```

使用 gcc 编译 signals.c，并生成可执行文件 signals：

```
#gcc -o signals signals.c
```

通过前面的介绍，读者已经熟悉，信号 SIGINT 由按键“Ctrl+C”发出，信号 SIGQUIT 由按键“Ctrl-\”发出。运行程序，得到输出结果：

```
# ./signals
process ID is 3116
Capture SIGINT signal, the signal number is 2(键入“Ctrl+c”)
Capture SIGQUIT signal, the signal number is 3(键入“Ctrl+\”)

[1]+  Stopped                    ./signals(键入“Ctrl+z”，进程置于后台)
# bg
[1]+ ./signals &
# kill -HUP 3116     /*向进程发送 SIGHUP 信号*/
Capture SIGHUP signal, the signal number is 1
# kill -9 3116       /*向进程发送 SIGKILL 信号，终止进程*/
```

读者可尝试自行分析程序代码的输出结果(其实已经在前文中详细讲解了)。

2. sigaction 函数

Linux 还提供了另外一种功能更为强大的信号处理机制——sigaction 系统调用。sigaction 函数的功能是检查或修改(或两者)与指定信号相关联的处理动作，此函数可以完全替代 signal 函数。sigaction 函数原型如下：

```
#include <signal.h>
int sigaction (int signum, const struct sigaction *act, struct sigaction *oldact);
```

返回：若成功则返回 0，若出错则返回-1。

参数 signum 为需要捕捉的信号。参数 act 是一个结构体，里面包含信号处理函数的地址、处理方式等信息。参数 oldact 是一个传出参数，sigaction 函数调用成功后，oldact 里面包含以前对 signum 信号的处理方式的信息。

结构体 struct sigaction(注意，名称与函数 sigaction 相同)的原型为：

```
struct sigaction
{
void (*sa_handler) (int);          /*老类型的信号处理函数指针*/
void (*sa_sigaction) (int, siginfo_t *, void *);/*新类型的信号处理函数指针*/
sigset_t sa_mask;                  /*将要被阻塞的信号集合*/
int sa_flags;                      /*信号处理方式掩码*/
void (*sa_restorer) (void);        /*保留，不使用*/
}
```

下面简要介绍 sigaction 结构体中的各字段含义及使用方式。

字段 sa_handler 是一个函数指针，用于指向原型为 void handler(int)的信号处理函数地址，即老类型的信号处理函数。

字段 sa_sigaction 也是一个函数指针，用于指向原型为：

```
void handler(int iSignNum, siginfo_t *pSignInfo, void *pReserved);
```

的信号处理函数，即新类型的信号处理函数。

该函数的 3 个参数含义为：

- iSignNum：传入的信号。
- pSignInfo：与该信号相关的一些信息，它是个结构体。
- pReserved：保留，未用。

注意，字段 sa_handler 和 sa_sigaction 应该只有一个生效，如果想采用老的信号处理机制，就应该让 sa_handler 指向正确的信号处理函数；否则应该让 sa_sigaction 指向正确的信号处理函数，并且让字段 sa_flags 包含 SA_SIGINFO 选项(稍后介绍)。

字段 sa_mask 是一个包含信号集合的结构体，该结构体内的信号表示在进行信号处理时，将要被阻塞的信号。针对 sigset_t 结构体，有一组专门的函数对它进行处理，稍后将进行介绍。

字段 sa_flags 指示了信号处理函数的不同选项，具体参数如表 9.1 所示。在实际使用时，通常是通过或运算串接不同的参数而实现所需的选项设置，将其赋值为 0 则表示选用所有的默认选项。

表 9.1　sa_flags 的取值及其含义

sa_flags 取值	含　义
SA_NOCLDSTOP	用于指定信号 SIGCHLD，当子进程被中断时，不产生此信号，当且仅当子进程结束时产生该信号
SA_NOCLDWAIT	对信号 SIGCHLD，当调用进程的子进程终止时，不创建僵死进程。若调用进程在后面调用 wait，则阻塞到它所有子进程都终止，此时返回-1，errno 设置为 ECHILD
SA_NODEFER	在处理信号时，如果又发生了其他的信号，则立即进入其他信号的处理，等其他信号处理完毕后，再继续处理当前的信号，即递规地处理。如果 sa_flags 包含了该选项，则结构体 sigaction 的 sa_mask 将无效
SA_NOMASK	同 SA_NODEFER 功能相似
SA_RESETHAND	处理完要捕捉的信号后，将自动撤销信号处理函数的注册，即必须再重新注册信号处理函数，才能继续处理接下来产生的信号。该选项不符合一般的信号处理流程，现已被废弃
SA_ONESHOT	同 SA_RESETHAND 功能相似
SA_RESTART	如果在发生信号时，程序正阻塞在某个系统调用，例如调用 read 函数，则在处理完毕信号后，接着从阻塞的系统返回。该选项符合普通的程序处理流程，一般应该设置该选项
SA_SIGINFO	指示结构体的信号处理函数指针是哪个有效，如果 sa_flags 包含该选项，则 sa_sigactiion 指针有效，否则是 sa_handler 指针有效

sigaction 函数不但可以实现 signal 函数的功能，而且还可以提供更加详细的信息，确切了解进程接收到信号时所发生的具体细节。程序 9.5 显示了 sigaction 函数的功能，当终端没有产生 SIGINT(Ctrl+c)或 SIGQUIT(Ctrl+\)信号时，程序能很好地执行 read 函数，即读入终端输入的字符串；当 SIGINT 或 SIGQUIT 信号产生时，进程被信号中断，read 出错退出了。代码如 sigaction.c。

【程序 9.5】sigaction 函数使用举例：sigaction.c。

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>

int g_iSeq=0;

void SignHandlerNew(int iSignNo,siginfo_t *pInfo,void *pReserved)
{
    int iSeq=g_iSeq++;
    printf("%d Enter SignHandlerNew,signo:%d.\n",iSeq,iSignNo);
    sleep(3);        /*睡眠 3 秒钟*/
    printf("%d Leave SignHandlerNew,signo:%d\n",iSeq,iSignNo);
}

int main(void)
{
char szBuf[20];        /*输入缓冲区，长度为 20*/
    int iRet;
    struct sigaction act;    /*包含信号处理动作的结构体*/
    act.sa_sigaction=SignHandlerNew;   /*指定信号处理函数*/
    act.sa_flags=SA_SIGINFO;    /*表明信号处理函数由 sa_sigaction 指定*/
    sigemptyset(&act.sa_mask);
/*信号集处理函数(稍后进行介绍)，将 act.sa_mask 所指向的信号集清空，*/
/*即不包含任何信号*/
sigaction(SIGINT,&act,NULL);    /*注册 SIGINT 信号*/
sigaction(SIGQUIT,&act,NULL);   /* 注册 SIGQUIT 信号*/
do{
iRet=read(STDIN_FILENO,szBuf,sizeof(szBuf)-1);   /*从标准输入读入数据*/
if(iRet<0)
{
perror("read fail.");
break;     /* read 出错退出*/
}
szBuf[iRet]=0;
printf("Get: %s",szBuf);      /*打印终端输入的字符串*/
}while(strcmp(szBuf,"quit\n")!=0);    /*输入“quit”时退出程序*/
return 0;
}
```

使用 gcc 编译 sigaction.c，并生成可执行文件 sigaction：

```
#gcc –o sigaction sigaction.c
```

运行程序，得到输出结果：

```
#./sigaction
hello,world!                          /*输入“hello,world!”*/
Get: hello,world!
I like linux C!                       /*输入“I like linux C!”*/
Get: I like linux C!
0 Enter SignHandlerNew,signo:2.       /*键入“Ctrl+c”，产生 SIGINT 信号*/
1 Enter SignHandlerNew,signo:3.       /*再次很快(间隔 3 秒钟之内)键入“Ctrl+\”，产生 SIGQUIT 信号*/
1 Leave SignHandlerNew,signo:3        /* SIGQUIT 信号处理完毕*/
0 Leave SignHandlerNew,signo:2        /* SIGINT 信号处理完毕*/
read fail.: Interrupted system call   /* 读出错，进程中断*/
#                                     /*程序退出(非正常)*/
```

通过程序运行的结果不难发现，当终端还没有产生 SIGINT 或 SIGQUIT 信号时，能正确地进行输入，并打印出输入的数据，而当信号产生时，进程被中断了。

再次运行程序，这一次不使用“Ctrl+c”或“Ctrl+\”来产生让进程终止的信号，而使用程序中设定的退出字符“quit”，这一次将不会看到读出错了，因为此时程序是正常退出的。如下所示：

```
#./sigaction
hello,world!                        /*输入“hello,world!”*/
Get: hello,world!
quit                                /*输入“quit”*/
Get: quit
#                                   /*程序退出(正常)*/
```

3. 信号集

在实际应用中，一个用户进程常常需要对多个信号进行出处理。为了方便同时对多个信号进行处理，在 Linux 系统中引入信号集(signal set)的概念。

信号集用于表示由多个信号所组成集合的数据类型，信号集定义为 sigset_t 类型的变量。POSIX.1 定义了下列 5 个处理信号集的函数：

```
#include <signal.h>
int sigemptyset (sigset_t *set);
int sigfillset (sigset_t *set);
int sigaddset (sigset_t *set, int signum);
int sigdelset (sigset_t *set, int signum);
```

4 个函数返回：若成功则返回 0，若出错则返回-1。

```
#include <signal.h>
int sigismember (const sigset_t *set, int signum);
```

返回：若为真则返回 1，若为假则返回 0。

其中，参数 set 是指向信号集的指针，参数 signum 用于表示一个信号。5 个函数的作用分别如下：

- sigemptyset 函数用于将 set 所指向的信号集设定为空，即不包含任何信号。
- sigfillset 函数用于将 set 所指向的信号集设定为满，即包含所有的信号。
- sigaddset 函数用于将 signum 所代表的信号添加到 set 所指向的信号集中。
- sigdelset 函数用于将 signum 所代表的信号从 set 所指向的信号集中删除。
- sigismember 函数用于检查 signum 所代表的信号是否在 set 所指向的信号集中。

一旦初始化了一个信号集，以后就可在该信号集中增、删特定的信号。对所有以信号集作为参数的函数，都向其传送信号集地址。在后面的学习中将经常使用到信号集。

例如，如果打算在处理信号 SIGINT 时，只阻塞对 SIGQUIT 信号的处理，可以用如下的方法(参考程序 9.5 中注册 SIGINT 和 SIGQUIT 信号的代码部分)：

```
struct sigaction act;
sigemptyset (&act.sa_mask);
sigaddset (&act.sa_mask, SIGQUIT);
sigaction (SIGINT, &act, NULL);
```

9.2.2 信号的发送

发送信号的函数有：kill、raise、sigqueue、alarm、setitimer 及 abort。下面分别介绍这些函数的具体含义和用法。

1. kill 函数

kill 函数用于向某一给定进程或进程组发送信号，函数原型如下：

```
#include <sys/types.h>
#include <signal.h>
int kill ( pid_t pid, int signum );
```

返回：若成功则返回 0，若出错则返回-1。

参数 pid 表示 kill 函数发送信号对象的进程或进程组号，其取值有几种情况，如表 9.2 所示。参数 signum 代表发送的信号。

表 9.2 pid 取值及对应含义

pid 取值	含　义
pid>0	将信号发送给进程号为 pid 的进程
pid=0	将信号发送给和目前进程相同进程组的所有进程
pid<0&& pid!=-1	向进程组 ID 为 pid 绝对值的进程组中的所有进程发送信号
pid=-1	除发送进程自身外，向所有进程 ID 大于 1 的进程发送信号(POSIX.1 未定义此种情况)

说　明

对于 pid<0 时的情况，哪些进程将接收信号，各种版本说法不一，其实很简单，参阅内核源码 kernel/signal.c 即可，上表中的规则是参考 Red Hat 9.0。

signum 是信号值，当为 0 时(即空信号)，实际不发送任何信号，但照常进行错误检查，因

此，可用于检查目标进程是否存在，以及当前进程是否具有向目标发送信号的权限(root 权限的进程可以向任何进程发送信号，非 root 权限的进程只能向属于同一个 session(会话)或者同一个用户的进程发送信号)。

kill 最常用于 pid>0 时的信号发送，调用成功返回 0；否则，返回-1。

程序 9.6 演示了父进程利用 kill 函数向其子进程传送一个 SIGABRT 信号，使子进程非正常结束，代码如 kill.c。

【程序 9.6】使用 kill 函数产生 SIGABRT 信号：kill.c。

```
#include<unistd.h>
#include<signal.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<stdio.h>
int main(void)
{
pid_t pid;
int status;
if(!(pid= fork()))
{
printf("Hi I am child process!\n");
sleep(10);              /*让子进程睡眠，看父进程的行为*/
printf("Hi I am child process, again!\n");
return 1;
}
else
{
printf("send signal to child process (%d) \n",pid);
sleep(1);
if(kill(pid ,SIGABRT)==-1)
{
   printf("kill failed!\n");
}
wait(&status);
if(WIFSIGNALED(status))
{
printf("child process receive signal %d\n",WTERMSIG(status));
           }
}
      return 0;
}
```

使用 gcc 编译 kill.c，并生成可执行文件 kill：

```
#gcc –o kill kill.c
```

运行程序，得到输出结果：

```
#./kill
Hi I am child process!
```

```
send signal to child process (2525)
child process receive signal 6
```

运行程序，从中可以看到，当父进程将信号 SIGABRT 发送给子进程(子进程 ID 为 2525)后，子进程非正常结束了——printf("Hi I am child process, again!\n");语句并没有执行。

2. raise 函数

raise 函数用于向进程本身发送信号，函数原型如下：

```
#include <sys/types.h>
#include <signal.h>
int raise (int signum);
```

返回：若成功则返回 0，若出错则返回-1。

参数 signum 为将要发送的信号值。

raise 函数的使用是很简单的，程序 9.7 利用 raise 函数向自身的进程发送了一个 SIGABRT 信号，这会使得进程非正常结束。代码如 raise.c。

【程序 9.7】使用 raise 函数产生 SIGABRT 信号：raise.c。

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    printf("Hello, I like Linux C programs!\n");
    if(raise(SIGABRT) == -1)      /*向进程本身发送 SIGABRT 信号*/
    {
            printf("raise failed!\n");      /*发送失败，退出*/
exit(1);
    }
    printf("Hello, I like Linux C programs,again!\n");      /*进程非正常结束，此句不会执行*/
    return 0;
}
```

使用 gcc 编译 raise.c，并生成可执行文件 raise：

```
#gcc –o raise raise.c
```

运行程序，得到输出结果：

```
#./raise
Hello, I like Linux C programs!
已放弃
#
```

显然，程序的运行结果正是我们所想要得到的。

3. sigqueue 函数

sigqueue 是比较新的发送信号系统调用，主要是针对实时信号提出的(当然也支持前 32 种)，

支持信号带有参数，通常与函数 sigaction 配合使用。函数原型如下：

```
#include<signal.h>
#include<unistd.h>
int sigqueue (pid_t pid, int signum, const union sigval val);
```

返回：若成功则返回 0，若出错则返回-1。

sigqueue 的第一个参数 pid 是指定接收信号的进程 ID，第二个参数 signum 确定即将发送的信号，第三个参数是一个联合数据结构 union sigval，指定了信号传递的参数，即通常所说的 4 字节值，定义如下：

```
typedef union sigval
{
int sival_int;
void *sival_ptr;        /*指向要传递的信号参数*/
}sigval_t;
```

sigqueue 比 kill 传递了更多的附加信息，但 sigqueue 只能向一个进程发送信号，而不能发送信号给一个进程组。如果 signum=0，将会执行错误检查，但实际上不发送任何信号，0 值信号可用于检查 pid 的有效性及当前进程是否有权限向目标进程发送信号。

在调用 sigqueue 时，sigval_t 指定的信息会复制到 3 参数信号处理函数(3 参数信号处理函数指的是信号处理函数由 sigaction 安装，并设定了 sa_sigaction 指针，在稍后的例子中读者可以清楚地看到)的 siginfo_t 结构中，这样信号处理函数就可以处理这些信息了。由于 sigqueue 系统调用支持发送带参数的信号，所以比 kill 系统调用的功能要灵活和强大得多。

注意

sigqueue 发送非实时信号时，第三个参数包含的信息仍然能够传递给信号处理函数；sigqueue 发送非实时信号时，仍然不支持排队，即在信号处理函数执行过程中到来的所有相同信号，都被合并为一个信号。

程序 9.8 演示了进程给自己发送信号 SIGUSR1，并且带上附加信息(一个字符串数据)。代码如 sigqueue.c。

【程序 9.8】使用 sigqueue 函数向进程自身发送 SIGUSR1 信号：sigqueue.c。

```
#include<stdio.h>
#include<signal.h>
#include<unistd.h>
#include<stdlib.h>
void SigHandler(int signo,siginfo_t *info,void *context)
{
    char *pMsg=(char*)info->si_value.sival_ptr;
    printf("Receive signal number:%d\n", signo);
    printf("Receive Message:%s\n", pMsg);
}
int main(void)
{
    struct sigaction sigAct;     /*定义包含信号处理动作的结构体*/
```

```
        sigAct.sa_flags=SA_SIGINFO;   /*表明信号处理函数由 sa_sigaction 指定*/
        sigAct.sa_sigaction=SigHandler;   /*指定信号处理函数*/
        if(sigaction(SIGUSR1,&sigAct,NULL)==-1)
        {
            printf("sigaction failed!\n");
            exit(1);
        }
        sigval_t val;            /*定义 sigqueue 函数的第三个参数*/
        char pMsg[ ]="I like Linux C programs!";
    /*将要传递的信息参数——一个字符串数据*/
        val.sival_ptr = pMsg;
        if(sigqueue(getpid(),SIGUSR1,val)==-1)
    /*调用 sigqueue 向自身发生 SIGUSR1 信号，并携带一个 4 字节的联合数据结构*/
        {
            printf("sigqueue failed!\n");
            exit(1);
        }
        sleep(3);
        return 0;
    }
```

使用 gcc 编译 sigqueue.c，并生成可执行文件 sigqueue：

```
#gcc –o sigqueue sigqueue.c
```

运行程序，得到输出结果：

```
#./sigqueue
Receive signal number:10
Receive Message: I like Linux C programs!
```

从中可以看到，进程成功接收到了自身发生的信号 10(SIGUSR1)，和信号的携带参数——字符串数据“I like Linux C programs!”。

4. alarm 函数

alarm 函数专门为 SIGALRM 信号而设，使系统在一定时间之后发送信号。函数原型如下：

```
#include<unistd.h>
unsigned int alarm (unsigned int seconds);
```

返回：如果调用 alarm 之前，进程中已经设置了闹钟时间，则返回上一个闹钟时间的剩余时间，否则返回 0。

参数 seconds 指定了下一次发送信号的时间，即在当期时间的 seconds 秒后，向进程本身发送 SIGALRM 信号，又称为闹钟时间。进程调用 alarm 后，任何以前的 alarm 调用都将无效。如果参数 seconds 为 0，那么进程内将不再包含任何闹钟时间。

程序 9.9 演示了 alarm 函数的用法，将 alarm 的时间参数设为 5 秒钟，5 秒钟之后将调用信号的处理函数。代码如 alarm.c。

【程序 9.9】使用 alarm 函数产生 SIGALRM 信号：alarm.c。

```
#include<unistd.h>
```

```
#include<signal.h>
#include <stdio.h>
void handler()
{
printf("Hello, I like Linux C programs!\n");
}
int main(void)
{
int i;
signal(SIGALRM,handler);
alarm(5);
for(i=1;i<7;i++)
{
printf("sleep %d ...\n",i);
sleep(1);
}
    return 0;
}
```

使用 gcc 编译 alarm.c，并生成可执行文件 alarm：

```
#gcc –o alarm alarm.c
```

运行程序，得到输出结果：

```
#./alarm
sleep 1 ...
sleep 2 ...
sleep 3 ...
sleep 4 ...
sleep 5 ...
Hello, I like Linux C programs!
sleep 6 ...
```

正如程序运行的结果所示，在 for 循环执行 5 次(说明过了大约 5 秒钟)之后，产生了 SIGALRM 信号，此时由 signal 注册信号的处理函数 handler，输出字符串。信号处理完毕后又返回先前程序的中断点，继续执行 for 循环。

5. setitimer 函数

setitimer 函数同 alarm 函数一样，也可以用于使系统在某一时刻发出信号，但它可以更加精确地控制程序。函数原型如下：

```
#include <sys/time.h>
int setitimer (int which, const struct itimerval *value, struct itimerval *oldvalue);
```

返回：若成功则返回 0，若出错则返回-1。

setitimer 的第一个参数 which 指定定时器类型，setitimer 比 alarm 功能强大，支持 3 种类型的定时器，如表 9.3 所示。参数 value 和 oldvalue 为指向时间参数的结构体指针，itimerval 结构原型如下：

```
struct itimerval
{
    struct timeval it_interval;   /*计时器重启动的间歇值*/
    struct timeval it_value;      /*计时器安装后首先启动的初始值*/
};
```

成员 it_interval 和 it_value 又是 timeval 类型的结构体：

```
struct timeval
{
    long tv_sec;              /*时间的秒数部分*/
    long tv_usec;             /*时间的微秒(1/1000000)部分*/
};
```

setitimer 将 value 指向的结构体设为计时器的当前值，如果 oldvalue 不是 NULL，将返回计时器原有值。

表 9.3　which 取值及对应定时器类型

which 取值	定时器类型	发 生 信 号
ITIMER_REAL	设定绝对时间，即根据系统的时间	SIGALRM
ITIMER_VIRTUAL	设定程序执行时间，只有在用户模式下才可跟踪时间	SIGVTALRM
ITIMER_PROF	从用户进程开始后开始计时	SIGPROF

程序 9.10 是关于 setitimer 系统调用的例子，代码如 setitimer.c 所示。

【程序 9.10】使用 setitimer 函数产生 SIGALRM 信号：setitimer.c。

```
#include <signal.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

static void ElsfTimer(int signo)          /*信号处理函数*/
{
   struct timeval tp;
   struct tm *tm;
   gettimeofday(&tp,NULL);     /* gettimeofday 函数获得系统当前时间(秒和微秒)*/
   tm=localtime(&tp.tv_sec);    /* localtime 取得当地目前时间和日期*/
   printf(" sec=%ld \t",tp.tv_sec);       /*打印从 UNIX 纪元开始到现在的秒数*/
   printf(" usec=%ld \n",tp.tv_usec);    /*打印微秒*/
   printf("%d-%d-%d%d:%d:%d\n",tm->tm_year+1900,tm->tm_mon+1,tm->tm_mday,
tm->tm_hour,tm->tm_min,tm->tm_sec);    /*打印当地目前时间和日期*/
}

static void InitTime(int tv_sec,int tv_usec)
{
   struct itimerval value;            /*定义时间参数结构体 value */
```

```
    signal(SIGALRM, ElsfTimer); /*注册信号 SIGALRM 和信号处理函数 ElsfTimer ()*/
    value.it_value.tv_sec=tv_sec;     /*秒*/
    value.it_value.tv_usec=tv_usec; /*微秒*/
    value.it_interval.tv_sec=tv_sec;
    value.it_interval.tv_usec=tv_usec;
    setitimer(ITIMER_REAL, &value, NULL);
/*setitimer 发送信号，定时类型为 ITIMER_REAL*/
}

int main(void)
{
    InitTime(5,0);          /*调用 InitTime 子函数，实参秒 tv_sec 为 5，微秒为 0*/
    while(1)                /*死循环，程序一直执行*/
    {
    }
    exit(0);
}
```

使用 gcc 编译 setitimer.c，并生成可执行文件 setitimer：

```
#gcc –o setitimer setitimer.c
```

运行程序，得到输出结果：

```
#./setitimer
sec=1255231656          usec=380650
2009 - 10 - 11 11:27:36
sec=1255231661          usec=380665
2009 - 10 - 11 11:27:41
sec=1255231666          usec=380517
2009 - 10 - 11 11:27:46
sec=1255231671          usec=380674
2009 - 10 - 11 11:27:51
sec=1255231676          usec=383250
2009 - 10 - 11 11:27:56
sec=1255231681          usec=380447
2009 - 10 - 11 11:28:1
sec=1255231686          usec=380186
2009 - 10 - 11 11:28:6
sec=1255231691          usec=385120
2009 - 10 - 11 11:28:11
sec=1255231696          usec=382144
2009 - 10 - 11 11:28:16
退出        /*键入“Ctrl+\”终止程序*/
```

从中可以看到，程序每隔 5 秒便会调用信号处理函数 ElsfTimer，打印出当前系统的时间和日期。需要说明的是，在ElsfTimer 函数中，使用了另外两个系统调用 gettimeofday 和 localtime。gettimeofday 的作用是获得以秒和微秒计时的系统的当前时间，这个时间是以 UNIX 操作系统的诞生之日开始计时的，我们一起来看看第一次的输出“sec=1255231656”，把它换算成年(1255231656÷3600÷24÷365=39.80)，大约是 39.80 年。是的，2009 年正是 UNIX 诞生 40 周

年。localtime 的作用是输出当地目前的时间和日期(关于这两个函数的介绍，已超出本书的范畴，读者可参考相关的 C 函数资料)。

提 示

程序 9.10 的执行结果中，第一行的输出：

```
sec = 1255231656          usec = 380650
```

意思为从UNIX 的诞生之时至该程序运行到此的时刻为 1255231656 秒又 380650 微秒。

另外需要说明的是，在程序 9.10 中，信号的处理函数完全没必要写的像 ElsfTimer 函数那样复杂，这里只是一个例子，笔者的目的是想让读者看到系统会每隔 5 秒钟输出当前的时间(当然，完全可以像 alarm.c 中那样使用 sleep(1);来表示一秒钟，尽管不是很精确)。

6. abort 函数

```
#include <stdlib.h>
void abort (void);
```

向进程发送 SIGABORT 信号，默认情况下进程会异常退出，当然可定义自己的信号处理函数。即使 SIGABORT 被进程设置为阻塞信号，调用 abort 后，SIGABORT 仍然能被进程接收。该函数无返回值。

abort 函数的使用与以上介绍的各个函数大同小异(甚至更简单)，这里不再举例赘述。

9.2.3 信号的阻塞

在 Linux 的信号控制中，有时候即不希望进程在接收到信号时立刻中断进程的执行，也不希望该信号完全被忽略，而是延迟一段时间再去调用相关的信号处理函数。这种操作就是通过阻塞信号的方法来实现的。信号阻塞的系统调用主要有 sigprocmask 和 sigsuspend 函数。

1. sigprocmask 函数

sigprocmask 函数可用于检测或更改(或两者)进程的信号掩码(signalmask)。信号掩码是由被阻塞的发送给当前进程的信号组成的信号集。

函数 sigaction 中设置的被阻塞信号集合只是针对于要处理的信号，例如：

```
struct sigaction act;
sigemptyset(&act.sa_mask);
sigaddset(&act.sa_mask,SIGQUIT);
sigaction(SIGINT,&act,NULL);
```

表示只有在处理信号 SIGINT 时，才阻塞信号 SIGQUIT。

而函数 sigprocmask 是全程阻塞，在 sigprocmask 中设置了阻塞集合后，被阻塞的信号将不能再被信号处理函数捕捉，直到重新设置阻塞信号集合。sigprocmask 的函数原型如下：

```
#include <signal.h>
int sigprocmask (int how, const sigset_t *set, sigset_t *oldset);
```

返回：若成功则返回 0，若出错则返回-1。

参数 set 和 oldset 是 sigset_t 类型的指针，用于表示所指向的信号集。set 指向一个信号集时，参数 how 表示 sigprocmask 函数将如何对 set 所指向信号集及信号掩码进行操作，其取值及对应函数功能见表 9.4。当 set 为 NULL 时，how 的取值无效。当 oldset 不为 NULL 时，函数 sigprocmask 将进程当前的信号掩码返回给 oldset。

表 9.4　how 的取值及对应函数功能

how 取值	对应函数功能
SIG_BLOCK	将 set 所指向的信号集中所包含的信号加到当前的信号掩码中，即信号掩码与 set 信号集做逻辑或运算
SIG_UNBLOCK	将 set 所指向的信号集中所包含的信号从当前的信号掩码中删除，即信号掩码与 set 信号集做逻辑减运算
SIG_SETMASK	设定新的当前信号掩码为 set 所指向的信号集中所包含的信号，即以 set 信号集对信号掩码进行赋值操作

将程序 9.5 进行部分修改后得到了程序 9.11，屏蔽掉 SIGINT 信号，那么信号处理函数 SigHandlerNew 将不能再捕捉 SIGINT(虽然注册了 SIGINT 和 SIGQUIT 两个信号)，而只能捕捉到 SIGQUIT 信号。代码如 block_sigint.c。

【程序 9.11】阻塞 SIGINT 信号：block_sigint.c。

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>

int g_iSeq=0;

void SignHandlerNew(int iSignNo,siginfo_t *pInfo,void *pReserved)    /*信号处理函数*/
{
      int iSeq=g_iSeq++;
      printf("%d Enter SignHandlerNew,signo:%d.\n",iSeq,iSignNo);
      sleep(3);          /*睡眠 3 秒钟*/
      printf("%d Leave SignHandlerNew,signo:%d\n",iSeq,iSignNo);
}

int main(void)
{
      char szBuf[20];        /*输入缓冲区，长度为 20*/
      int iRet;
      struct sigaction act;      /*包含信号处理动作的结构体*/
      act.sa_sigaction=SignHandlerNew;      /*指定信号处理函数*/
      act.sa_flags=SA_SIGINFO;      /*表明信号处理函数由 sa_sigaction 指定*/
      /*屏蔽掉 SIGINT 信号，SigHandlerNew 将不能再捕捉 SIGINT*/
sigset_t sigSet;
      sigemptyset(&sigSet);
      sigaddset(&sigSet,SIGINT);
```

```
        sigprocmask(SIG_BLOCK,&sigSet,NULL);
        sigemptyset(&act.sa_mask);
        sigaction(SIGINT,&act,NULL);     /*注册 SIGINT 信号*/
        sigaction(SIGQUIT,&act,NULL);    /*注册 SIGQUIT 信号*/
        do{
            iRet=read(STDIN_FILENO,szBuf,sizeof(szBuf)-1);   /*从标准输入读入数据*/
            if(iRet<0){
                perror("read fail.");
                break;      /* read 出错退出*/
            }
            szBuf[iRet]=0;
            printf("Get: %s",szBuf);            /*打印终端输入的字符串*/
        }while(strcmp(szBuf,"quit\n")!=0);   /*输入"quit"时退出程序*/
        return 0;
    }
```

使用 gcc 编译 block_sigint.c，并生成可执行文件 block_sigint：

```
#gcc –o block_sigint block_sigint.c
```

运行程序，得到输出结果：

```
#./block_sigint
hello,world!                          /*输入"hello,world!"*/
Get: hello,world!
I like linux C!                       /*输入"I like linux C!"*/
Get: I like linux C!
0 Enter SignHandlerNew,signo:3.       /*键入"Ctrl+\"，产生 SIGQUIT 信号*/
0 Leave SignHandlerNew,signo:3        /* SIGQUIT 信号处理完毕*/
read fail.: Interrupted system call   /*读出错，进程中断*/
#                                     /*程序退出(非正常)*/
```

与程序 9.5 的运行结果相比，当终端键入“Ctrl+c”时，进程不再有反应，原因如前所述，有“Ctrl+c”产生的 SIGINT 信号被 sigprocmask 函数设置为屏蔽了。

2. sigsuspend 函数

sigsuspend 函数用于使进程挂起，在调用 sigsuspend 后，进程就挂起在那里，等待着开放的信号的唤醒。系统在接收到信号后，马上就把现在的信号集还原为原来的，然后调用处理函数。函数原型如下：

```
#include <signal.h>
int sigsuspend (const sigset_t *sigmask);
```

返回：若出错则返回-1，errno 设置为 EINTR。

进程的信号屏蔽字设置为由参数 sigmask 指向的值。在捕捉到一个信号或发生了一个会终止该进程的信号之前，该进程也被挂起。如果捕捉到一个信号而且从该信号处理程序返回，则 sigsuspend 返回，并且该进程的信号屏蔽字设置为调用 sigsuspend 之前的值。

注意，此函数没有成功返回值。如果它返回到调用者，则总是返回-1，并且 errno 设置为 EINTR(表示一个被中断的系统调用)。

9.2.4 计时器与信号

在实际的编程中，经常会使用 Linux 下基于定时机制的信号，因为这种信号使得应用程序更加简洁和稳定，也为程序员节省了不少工作量。在前面的内容中，读者已经多次见到了这种函数的使用，但这里有必要再进行一些补充。

1. 睡眠函数

Linux 下有两个睡眠函数，原型为：

```
#include <unistd.h>
unsigned int sleep (unsigned int seconds);
void usleep (unsigned long usec);
```

函数 sleep 让进程睡眠 seconds 秒，函数 usleep 让进程睡眠 usec 毫秒。

事实上，sleep 睡眠函数的内部是用信号机制进行处理的，用到的函数有：

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
int pause(void);
```

alarm 函数告知自身进程，使进程在 seconds 秒后自动产生一个 SIGALRM 信号，这在 9.2.2 小节中已经详细介绍了。而 pause 函数用于将自身进程挂起，直到有信号发生时才从 pause 返回。

程序 9.12 是一个很简单的例子，使进程模拟睡眠 3 秒钟。代码如 pause.c。

【程序 9.12】使用 pause 函数将进程挂起：pause.c。

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void SignHandler(int iSignNo)
{
    printf("signal:%d\n",iSignNo);
}
int main(void)
{
    signal(SIGALRM,SignHandler);
    alarm(3);
    printf("Before pause().\n");
    pause();
    printf("After pause().\n");
    return 0;
}
```

使用 gcc 编译 pause.c，并生成可执行文件 pause：

```
#gcc –o pause pause.c
```

运行程序，得到输出结果：

```
#./pause
```

```
Before pause().
signal:14
After pause().
```

读者可自己在 Linux 系统下运行这个程序。可以看到，除了进程成功捕捉到信号 SIGALRM 外，在程序执行结果的第一行输出“Before pause().”之后，大约相隔了 3 秒钟，才输出了第二行“signal:14”。

提 示

因为 sleep 在内部是用 alarm 来实现的，所以在程序中最好不要使 sleep 与 alarm 混用，以免造成混乱。

2. 时钟处理

Linux 系统为每个进程维护 3 个计时器，分别是真实计时器、虚拟计时器和实用计时器。

- 真实计时器计算的是程序运行的实际时间。
- 虚拟计时器计算的是程序运行在用户态时所消耗的时间(可认为是实际时间减掉系统调用和程序睡眠所消耗的时间)。
- 实用计时器计算的是程序处于用户态和处于内核态所消耗的时间之和。

例如：有一程序运行，在用户态运行了 5 秒，在内核态运行了 6 秒，还睡眠了 7 秒，则真实计算器计算的结果是 18 秒，虚拟计时器计算的是 5 秒，实用计时器计算的是 11 秒。

用指定的初始间隔和重复间隔时间为进程设定好一个计时器后，该计时器就会定时地向进程发送时钟信号。3 个计时器发送的时钟信号分别为：SIGALRM、SIGVTALRM 和 SIGPROF。

用到的函数有 getitimer 和 setitimer。函数原型如下：

```
#include <sys/time.h>
int getitimer(int which, struct itimerval *value);
int setitimer(int which, const struct itimerval *value, struct itimer val *ovalue);
```

返回：若成功则返回 0，若出错则返回-1。

getitimer 用于获取计时器的设置。参数 which 用于指定计时器的类型，可选项为 ITIMER_REAL(真实计时器)、ITIMER_VITUAL(虚拟计时器)、ITIMER_PROF(实用计时器)。value 为一结构体的传出参数，用于传出该计时器的初始间隔时间和重复间隔时间

setitimer 用于设置计时器，在 9.2.2 小节中已经进行详细讲解了，这里不再赘述。

9.3 本章小结

信号是系统中用于处理异步事件的主要手段。本章主要介绍了 Linux 下信号的基本概念、信号处理的机制及信号操作的一些相关系统调用。这些系统调用包括信号处理函数、信号发送函数、信号阻塞函数等，此外还补充了 Linux 下常见的计时器相关函数。信号的使用对于灵活使用 C 语言在 Linux 环境下进行程序开发是非常有益的，事实上，在编写大型的程序时，经常会需要处理多个进程之间的异步事件，所以是离不开信号的使用的。

实战演练

1. 使用“kill -l”命令查看 Linux 系统中的信号列表。
2. 编写一个程序，使用 signal 函数捕捉从终端键入“Ctrl+\”时产生 SIGQUIT 信号，获得该信号的信号值。
3. 编写一个程序，使用 signal 函数忽略从终端键入“Ctrl+\”时产生 SIGQUIT 信号。
4. 编写一个程序，使用 sigaction 函数捕捉从终端键入“Ctrl+\”时产生 SIGQUIT 信号，以及从终端键入“Ctrl+c”时产生的 SIGINT 信号。
5. 编写一个程序，在父进程中使用 kill 函数向其子进程发送一个 SIGABRT 信号，使子进程非正常结束。
6. 编写一个程序，使用 raise 函数向进程自身发送一个 SIGABRT 信号，使进程非正常结束。
7. 编写一个程序，使用 sigqueue 函数向进程自身发送一个 SIGUSR1 信号，并获取该信号的信号值。
8. 编写一个程序，使用 alarm 函数产生 SIGALRM 信号，将 alarm 的时间参数设为 3 秒钟，在此期间获得当前进程的 ID 并打印输出。
9. 编写一个程序，使用 sigprocmask 函数阻塞从终端键入“Ctrl+c”时产生的 SIGINT 信号。
10. 编写一个程序，使用 pause 函数将进程挂起，直到有 SIGALRM 信号发生时才从 pause 返回。

第10章

进程间通信

Linux 作为一个多任务多进程的操作系统，各个进程之间的信息交互是不可避免的。进程间通信就是要在不同的进程之间传播或交换信息。另外，进程间通信也是现代程序设计的重要手段，借助于进程间通信的使用，可以建立分布式的应用程序。Linux 系统为立分布式的应用程序提供了强大而方便的实现平台，甚至可以将整个万维网看作是分布式的应用程序。

进程间的通信可以分为本地进程间通信和远程进程间通信。本章将向读者介绍本地进程间通信的实现机制，远程进程间通信将在第 12 章“网络编程”中介绍。



本章内容：

◎ 进程间通信简介。
◎ 管道。
◎ 命名管道。
◎ 消息队列。
◎ 共享内存。
◎ 信号量。

10.1 进程间通信简介

进程间通信(InterProcess Communication，IPC)就是在不同进程之间传播或交换信息，那么不同进程之间存在着什么样的能使双方都可以访问的介质呢？进程的用户空间是互相独立的，一般而言是不能互相访问的，唯一的例外是共享内存区。由于系统空间是一个“公共场所”，所以内核显然可以提供共享内存的条件。

除此以外，那就是双方都可以访问的外设了。在这个意义上，两个进程当然也可以通过磁盘上的普通文件交换信息，或者通过“注册表”或其他数据库中的某些表项和记录交换信息。广义上讲，这也是进程间通信的手段，但是一般都不把它算做“进程间通信”。因为这样的通信手段效率实在太低了，而人们对进程间通信的要求是要有一定的实时性。

Linux 的进程间通信的方法有管道、消息队列、信号量、共享内存、套接口等。其中，管道又分为命名管道和无名管道。消息队列、信号量、共享内存通称为系统(POSIX 和 System V 系统)IPC。管道、消息队列、信号量和共享内存用于本地进程间通信，而套接口用于远程进程间通信。下面进行简单介绍。

- 管道(Pipe)及命名管道(named pipe)：管道可用于具有亲缘关系进程间的通信，命名管道克服了管道没命名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。
- 消息(Message)队列(报文队列)：消息队列是消息的链接表，包括 POSIX 消息队列 System V 消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读取队列中的消息。消息队列克服了信号量承载信息量少，管道只能承载无格式字节流及缓冲区大小受限等缺点。
- 共享内存：使得多个进程可以访问同一块内存空间，是最快的可用 IPC 形式。是针对其他通信机制运行效率较低而设计的。往往与其他通信机制，如信号量结合使用，来达到进程间的同步及互斥。
- 信号量(semaphore)：主要作为进程间及同一进程不同线程之间的同步手段。
- 套接口(Socket)：也称套接字，是更为一般的进程间通信机制，可用于不同机器之间(远程)的进程间通信。起初是由 UNIX 系统的 BSD 分支开发出来的，但现在一般可以移植到其他类 UNIX 系统上：Linux 和 System V 的变种都支持套接字。

说 明

事实上，也可以将第 9 章中介绍的信号(用于通知接收进程有某种事件发生)归结为进程间的通信方式之一。应该提醒读者注意的是，信号和信号量是有区别的，甚至可以说两者是截然不同的，千万不能把它们混为一谈。信号(signal)是一种处理异步事件的方法。信号是由硬件或软件触发，再由操作系统内核发送给应用程序的中断形式。POSIX 定义了一系列的信号集。信号量(semaphore)是一种实现进程间同步、互斥的机制。信号量是 POSIX 进程间通信的工具，在它上面定义了一系列操作原语，简单地讲它可以在进程间进行通信。

进程间通信就是让多个进程之间可以互相访问，这种访问包括程序运行的适时数据，也包括对方的代码段，这是在实际应用中及其常见的问题，进程间通信示意图如图 10.1 所示。

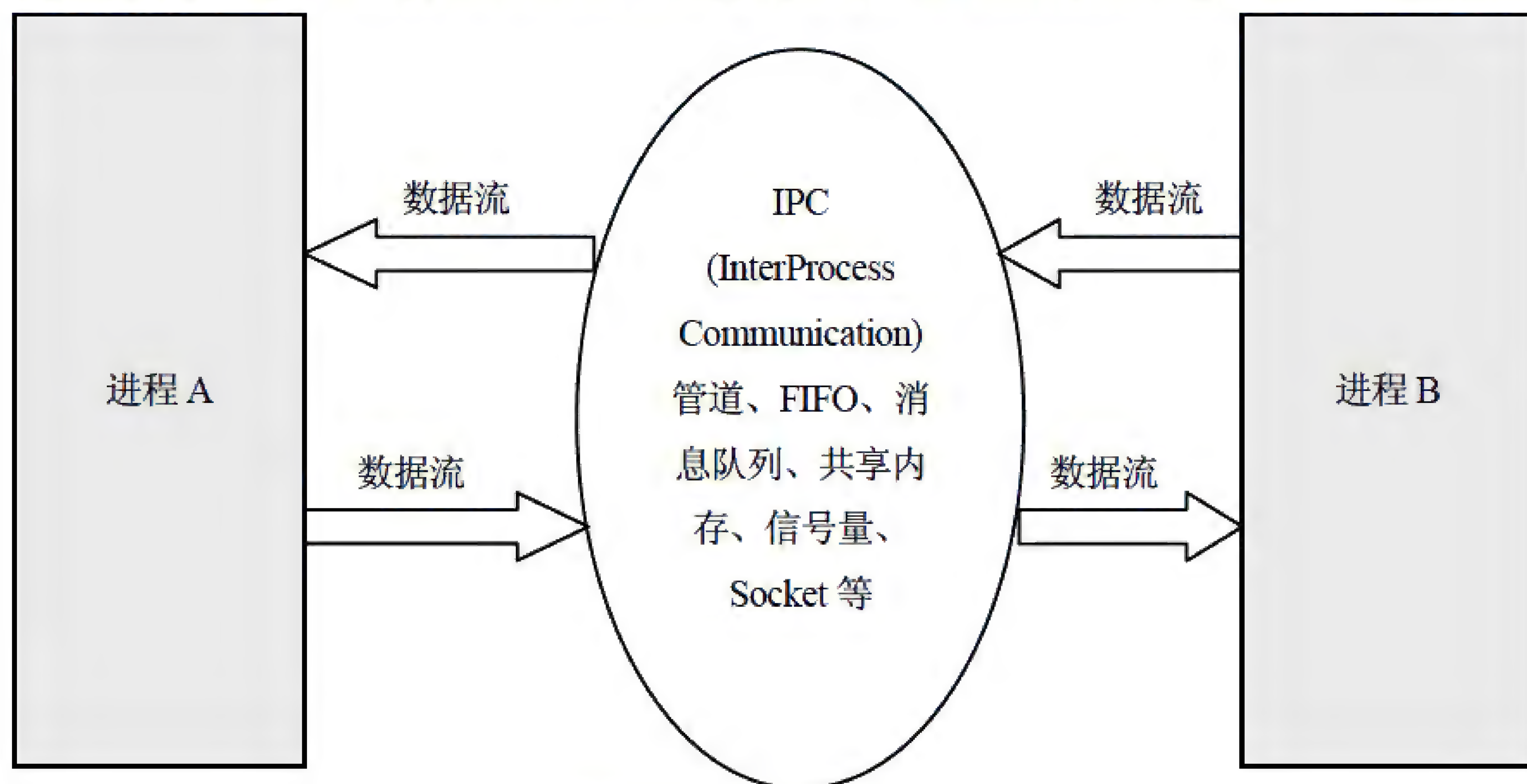


图 10.1　进程间通信

上图所示进程间通信的模式中，进程 A 和进程 B 在运行的过程中会需要一些外部的数据，IPC 为两个进程提供了一种数据传输的通道。

10.2 管道

管道(Pipe)，也称为匿名管道，是 Linux 下最常见的进程间通信方式之一，它是在两个进程之间实现一个数据流通的通道。管道是一种很经典的进程之间的通信方式，其优点在于简单易用，其缺点在于功能简单，有很多限制。本节主要介绍管道的相关操作。

10.2.1　管道的概念

管道是 Linux/UNIX 系统中比较原始的进程间通信形式，它实现数据以一种数据流的方式在进程间流动。在系统中其相当于文件系统上的一个文件，来缓存所要传输的数据。在某些特性上又不同于文件，例如，当数据读出后，则管道中就没有数据了，但文件没有这个特性。

顾名思义，匿名管道在系统中是没有实名的，并不可以在文件系统中以任何方式看到该管道。它只是进程的一种资源，会随着进程的结束而被系统清除。创建一个管道时生成了两个文件描述符(稍后将看到)，但对于管道中所使用的文件描述符并没有路径名，也就是不存在任何意义上的文件，它们只是在内存中与某一个索引节点相关联的两个文件描述符。

管道通信是在 Linux 系统中应用比较频繁的一种方式，尤其是在 Shell 中，经常使用管道(|)来连接两个甚至多个命令。

例如，回忆我们在 9.1.1 小节中用到的“kill -l”命令，它打印了当前系统的信号列表，如果想在该信号列表中直接查找含有字串“SIGRTMIN”的命令，可以使用管道(|)来连接“kill -l”和“grep”命令，实际上是 Shell 程序创建了 kill -l 和 grep 两个进程和这两个进程间的管道。下

面是笔者在 Ubuntu 12.04 系统下的运行结果(读者可以对照 9.1.1 小节中的示例进行分析)：

```
kill -l|grep SIGRTMIN
31) SIGSYS          34) SIGRTMIN        35) SIGRTMIN+1      36) SIGRTMIN+2      37) SIGRTMIN+3
38) SIGRTMIN+4      39) SIGRTMIN+5      40) SIGRTMIN+6      41) SIGRTMIN+7      42) SIGRTMIN+8
43) SIGRTMIN+9      44) SIGRTMIN+10     45) SIGRTMIN+11     46) SIGRTMIN+12     47) SIGRTMIN+13
48) SIGRTMIN+14     49) SIGRTMIN+15     50) SIGRTMAX-14     51) SIGRTMAX-13     52) SIGRTMAX-12
```

另外，上述命令中使用的是半双工管道，即 kill -l 命令的输出是 grep 命令的输入(而不能是：grep 命令的输出是 kill -l 命令的输入)。管道从数据流动方向上又分全双工管道及半双工管道，当然全双工管道现在某些系统还不支持，其在具体的实现过程中也只是在文件打开的方式上有一些区别(在操作规则上也有一些不同，全双工管道要相比半双工复杂得多)。

管道通信具有以下特点：

- 管道没有名字，所以也称为匿名管道。
- 管道是半双工的(至少在大多数系统中是这样的)，数据只能向一个方向流动。需要双方通信时，需要建立起两个管道。
- 只能用于父子进程或者兄弟进程之间(具有亲缘关系的进程)。
- 单独构成一种独立的文件系统。管道对于管道两端的进程而言，就是一个文件，但它不是普通的文件，它不属于某种文件系统，而是自立门户，单独构成一种文件系统，并且只存在于内存中。
- 数据的读出和写入：一个进程向管道中写的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。
- 管道的缓冲区是有限的(管道制只存在于内存中，在管道创建时，为缓冲区分配一个页面大小)。
- 管道所传送的是无格式字节流，这就要求管道的读出方和写入方必须事先约定好数据的格式，比如多少字节算做一个消息(或命令，或记录)等。

下面介绍关于管道操作的具体调用。

10.2.2 管道的创建与关闭

Linux 环境下使用 pipe 函数创建一个匿名管道，其函数原型如下：

```
#include <unistd.h>
int pipe (int fd[2]);
```

返回：若成功则返回 0，若出错则返回-1。

参数 fd[2]是一个长度为 2 的文件描述符数组，fd[0]是读出端的文件描述符，fd[1]是写入端的文件描述符。当函数成功返回后，则自动维护了一个从 fd[1]到 fd[0]的数据通道。

管道的关闭使用的是基于文件描述符的 close 函数(参考 6.2.4 小节)。

程序 10.1 演示了如何使用 pipe 函数创建管道及关闭管道。程序中首先使用函数 pipe 建立管道，并使用管道传输数据，在程序的结束部分，释放掉管道占用的文件资源(两个文件描述符)，代码如 create_pipe.c。

【程序 10.1】创建一个匿名管道：create_pipe.c。

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int fd[2];                    /* 管道的文件描述符数组 */
    char str[256];
    if((pipe(fd)) < 0)
        {
        printf("create the pipe failed!\n");
        exit(1);        /*pipe 出错退出*/
     }
    write(fd[1], "create the pipe successfully!\n", 31 );    /*向管道写入端写入数据*/
    read(fd[0], str, sizeof(str) );    /*从管道读出端读出数据*/
    printf ("%s", str );
    printf ("pipe file descriptors are%d,%d \n", fd[0], fd[1]) ;
    close(fd[0]);                 /* 关闭管道的读出端文件描述符*/
    close(fd[1]);                 /* 关闭管道的写入端文件描述符*/
    return 0;
}
```

使用 gcc 编译 create_pipe.c，并生成可执行文件 create_pipe：

```
#gcc –o create_pipe create_pipe.c
```

运行程序，得到输出结果：

```
#./ create_pipe
create the pipe successfully!
pipe file descriptors are 3,4
```

程序中使用 pipe 函数成功建立了一个匿名管道 fd。

另外，文件描述符数组 fd 并没有和任何有名文件相关联，之后向管道一端写入数据并从读出端读出数据，将数据输出到标准输出。在程序的最后使用 close 函数关闭管道的两端。

10.2.3 管道的读写

可以使用 read 和 write 函数对管道进行读写操作，需要注意的是，管道的两端是固定了任务的，即管道的读出端只能用于读取数据，管道的写入端则只能用于写入数据。如果试图从管道写端读取数据，或者向管道读端写入数据都将导致错误发生。一般文件的 I/O 函数都可以用于管道，如 close、read、write 等。

当对一个读端已经关闭的管道进行写操作时，会产生信号 SIGPIPE，说明管道读端已经关闭，并且 write 操作返回为-1，errno 的值设为 EPIPE，对于 SIGPIPE 信号可以进行捕捉处理。如果写入进程不能捕捉或者干脆忽略 SIGPIPE 信号，则写入进程会中断。

管道的读取规则是：如果管道的写端不存在，则认为已经读到了数据的末尾，读函数返回的读出字节数为 0；当管道的写端存在时，如果请求的字节数目大于 PIPE_BUF，则返回管道中现有的数据字节数，如果请求的字节数目不大于 PIPE_BUF，则返回管道中现有数据字节数

(此时，管道中数据量小于请求的数据量)；或者返回请求的字节数(此时，管道中数据量不小于请求的数据量)。

说 明

PIPE_BUF 在 include/linux/limits.h 中定义，不同的内核版本可能会有所不同。POSIX.1 要求 PIPE_BUF 至少为 512 字节，Ubuntu 12.04 中为 4096。

另外，在进行读写管道时，对一个管道进行读操作后，read 函数返回为 0，有两种意义，一种是管道中无数据并且写入端已经关闭(即写端不存在)。另一种是管道中无数据，写入端依然存活。这两种情况要根据需要分别处理。

管道的写入规则是：向管道中写入数据时，管道缓冲区一旦有空闲区域，写进程就会立即试图向管道写入数据。如果读进程未读取管道缓冲区中的数据，那么写操作将一直阻塞。

提 示

只有在管道的读端存在时，向管道中写入数据才有意义。否则，向管道中写入数据的进程将收到内核传来的 SIFPIPE 信号，应用程序可以处理该信号，也可以忽略(默认动作则是应用程序终止)。

从实例程序 10.1 中可以发现，单独一个进程操作管道是没有任何意义的，管道的应用一般体现在父子进程或者兄弟进程的通信。

如果要建立一个父进程到子进程的数据通道，可以先调用 pipe 函数紧接着调用 fork 函数，由于子进程自动继承父进程的数据段，则父子进程同时拥有管道的操作权，此时管道的方向取决于用户怎么维护该管道，管道示意图如图 10.2 所示。

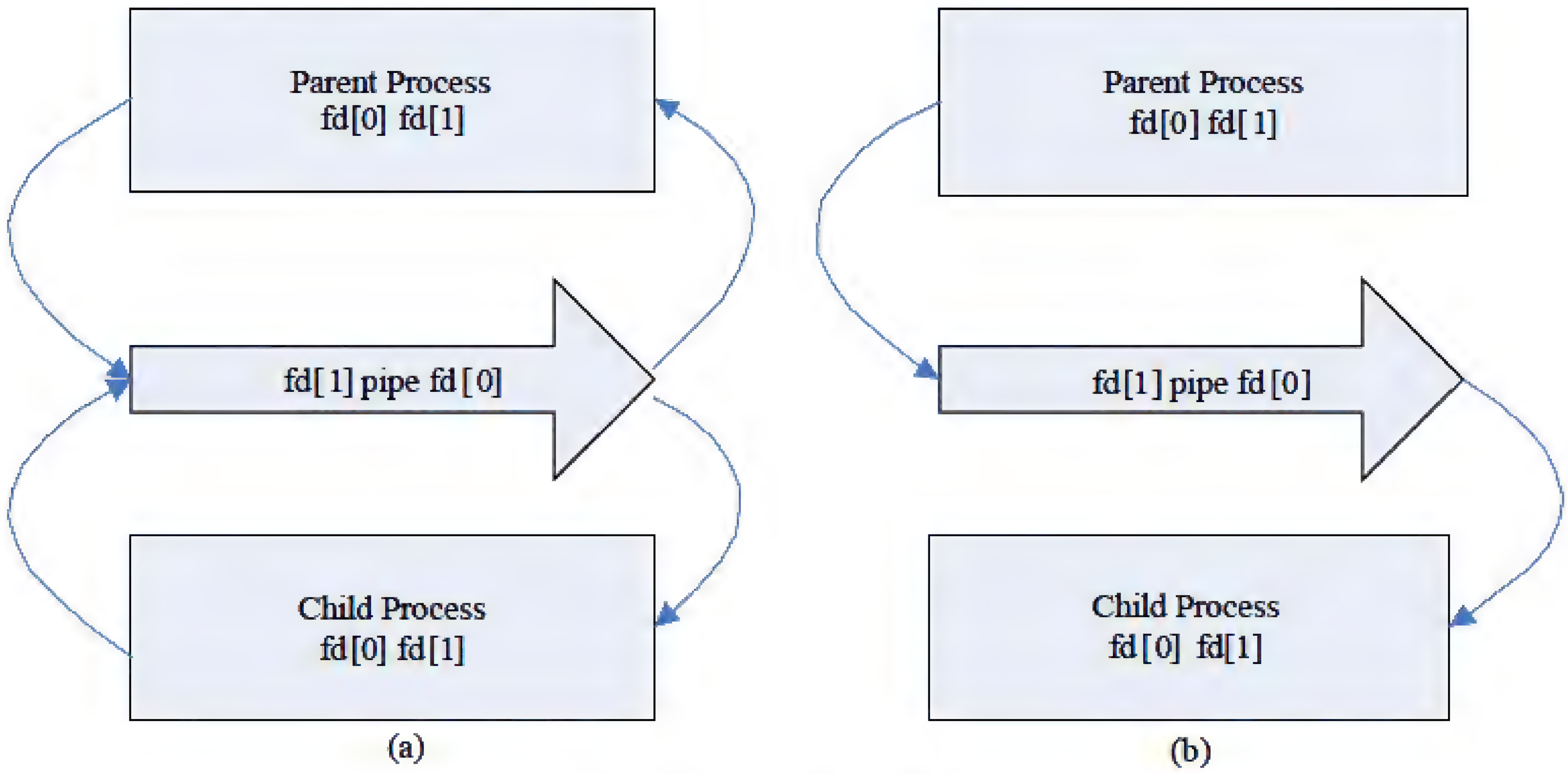


图 10.2　管道示意图

当用户想要建立一个父进程到子进程的数据通道(即数据传送方向是从父进程到子进程)时，也就是要在子进程中读出之前在父进程中写入的数据，那么就要在父进程中关闭管道的读出端(以顺利地写入数据)，相应地在子进程中关闭管道的写入端(以顺利地读出数据)，如图 10.2(b)所示。相反的，当维护子进程到父进程的数据通道时，在父进程中关闭写入端，子进程中关闭读出端即可。总之，使用 pipe 及 fork 组合，可以构造出所有的父进程与子进程，或子进程到兄弟进程的管道。

程序 10.2 是关于管道在父子进程中的应用示例，演示了使用 pipe 及 fork 组合实现父子进程通信。程序中先使用 pipe 函数建立管道，再使用 fork 函数创建子进程。在父子进程中维护管道的数据方向，并在父进程中向子进程发送消息，在子进程中接收消息并输出到标准输出。程序清单见 parent_pipe_child.c。

【程序 10.2】父进程利用管道向子进程发送消息：parent_pipe_child.c。

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#define BUFSIZE PIPE_BUF    /* PIPE_BUF：管道默认一次性读写的数据长度*/

void err_quit(char *msg)
{
    printf (msg);
    exit(1);
}

int main(void)
{
    int fd[2];
    char buf[BUFSIZE]="hello my child!\n";     /*写入管道的缓冲区*/
    pid_t pid;
    int len;
    if ((pipe(fd))<0)     /*创建管道*/
{
        err_quit ("pipe failed\n");
    }
    if ((pid = fork())<0)   /*创建一个子进程*/
{
        err_quit ("fork failed\n");
    }
    else if (pid>0)
{
        close ( fd[0] );                    /*父进程中关闭管道的读出端*/
        write (fd[1], buf, strlen(buf));    /*父进程向管道写入数据*/
        exit (0);
    }
    else
{
        close ( fd[1] );                    /*子进程关闭管道的写入端*/
        len=read (fd[0], buf, BUFSIZE);     /*子进程从管道中读出数据*/
        if (len < 0)
{
            err_quit ("process failed when read a pipe\n");
        }
      else
```

```
            {
                write(STDOUT_FILENO, buf, len);    /*输出到标准输出*/
            }
        exit(0);
        }
}
```

使用 gcc 编译 parent_pipe_child.c，并生成可执行文件 parent_pipe_child：

```
#gcc –o parent_pipe_child parent_pipe_child.c
```

运行程序，得到输出结果：

```
#./ parent_pipe_child
hello my child!
```

程序中使用 pipe 函数加 fork 组合，实现父进程到子进程的通信。程序在父进程段中关闭了管道的读出端，并相应地在子进程中关闭了管道的输入端，从而实现数据从父进程流向子进程。

而管道在兄弟进程间应用时，应该先在父进程中建立管道，然后调用 fork 函数创建两个子进程，在兄弟子进程中维护管道的数据方向。

说 明

这里的问题是维护管道的顺序，当父进程创建了管道，只有子进程已经继承了管道后，父进程才可以执行关闭管道的操作。如果在 fork 之前已经关闭管道，子进程将不能继承任何可以使用的管道。

程序 10.3 是关于管道在兄弟进程间的应用实例，演示了管道在兄弟进程间通信。下例中在父进程中创建管道，并使用 fork 函数创建两个子进程。在第 1 个子进程中发送消息到第 2 个子进程，第 2 个子进程中读出消息并处理。在父进程中，由于并不使用管道通信，所以什么都不做，直接关闭管道的两端并退出。程序清单见 brother_pipe.c。

【程序 10.3】管道在兄弟进程间传递数据：brother_pipe.c。

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <limits.h>
#include <string.h>
#define BUFSIZE PIPE_BUF    /* PIPE_BUF：管道默认一次性读写的数据长度*/

void err_quit(char * msg)
{
    printf ( msg );
    exit(1);
}

int main(void)
```

```
{
    int fd[2];
    char buf[BUFSIZE]="hello my brother!\n";            /* 缓冲区 */
    pid_t pid;
    int len;
    if ( (pipe(fd))<0 )                  /*创建管道*/
      {
          err_quit("pipe failed\n");
      }
    if ( (pid = fork())<0 )              /*创建第一个子进程*/
      {
          err_quit("fork failed\n");
      }
    else if ( pid==0 )                   /*子进程中*/
{
          close ( fd[0] );                       /*关闭不使用的文件描述符*/
          write(fd[1], buf, strlen(buf));        /*写入消息*/
          exit(0);
      }
    if ( (pid = fork())<0 )                      /*创建第二个子进程*/
      {
          err_quit("fork failed\n");
      }
    else if ( pid>0 )                    /*父进程中*/
{
          close ( fd[0] );
          close ( fd[1] );
          exit ( 0 );
      }
    else                                 /*第二个子进程中*/
{
          close ( fd[1] );                          /*关闭不使用的文件描述符*/
          len=read (fd[0], buf, BUFSIZE);           /*读取消息*/
          write(STDOUT_FILENO, buf, len);           /*将消息输出到标准输出*/
          exit(0);
      }
    return 0;
}
```

使用 gcc 编译 brother_pipe.c，并生成可执行文件 brother_pipe：

```
#gcc –o brother_pipe brother_pipe.c
```

运行程序，得到输出结果：

```
#./ brother_pipe
hello my brother!
```

上述程序中父进程分别建立了两个子进程，在子进程 1 中关闭了管道的读出端，在子进程 2 中关闭了管道的输入端，并在父进程中关闭了管道的两端。

另外，程序中父进程在创建第 1 个子进程时并没有关闭管道两端，而是在创建第 2 个进程

时才关闭管道。这是为了在创建第 2 个进程时，子进程可以继承存活的管道，而不是一个两端已经关闭的管道。

10.3 命名管道

命名管道(named pipe)也称为 FIFO，它是一种文件类型，在文件系统中可以看到它(在 6.1.2 小节中提到过管道文件)，创建一个 FIFO 文件类似于创建一个普通文件。在程序中可以通过查看文件 stat 结构中 st_mode 成员的值来判断该文件是否为FIFO(stat 结构体在 6.4 节进行了阐述)。本节向读者介绍 FIFO 管道。

10.3.1 命名管道的概念

管道应用的一个重大限制是它没有名字，因此，只能用于具有亲缘关系的进程间通信，在命名管道(或 FIFO)提出后，该限制得到了克服。FIFO 不同于管道之处在于它提供一个路径名与之关联，以 FIFO 的文件形式存在于文件系统中。这样，即使与 FIFO 的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能够彼此通过 FIFO 相互通信(能够访问该路径的进程及 FIFO 的创建进程之间)，因此，通过 FIFO 不相关的进程也能交换数据。

归纳起来，命名管道区别于管道主要体现在以下两点：

- 命名管道可以用于任何两个进程间的通信，而并不限制这两个进程同源，因此命名管道的使用比管道的使用要灵活方便得多。
- 命名管道作为一种特殊的文件存放于文件系统中，而不是像管道一样存放于内存(使用完毕后消失)。当进程对命名管道的使用结束后，命名管道依然存在于文件系统中，除非对其进行删除操作，否则该命名管道不会消失。

FIFO 的出现，也极好地解决了系统在应用过程中产生的大量的中间临时文件的问题。FIFO 可以被 Shell 调用使数据从一个进程到另一个进程，系统不必为该中间通道而清理不必要的垃圾，或者去释放该通道的资源，它可以被留作后来的进程使用。

另外，需要注意的是，FIFO 严格遵循先进先出(first in first out)的规则，对管道及 FIFO 的读总是从开始处返回数据，对它们的写则把数据添加到末尾。它们不支持诸如 lseek 等文件定位操作。

在 Shell 中可以使用 mkfifo 命令建立一个命名管道，mkfifo 命令的格式如下所示：

```
mkfifo [option] name...
```

其中 option 选项中可以选择要创建 FIFO 的模式，使用形式为-m mode，这里 mode 指出将要创建 FIFO 的八进制模式，注意，这里新创建的 FIFO 会像普通文件一样受到创建进程的 umask 修正。name 表示所要创建的 FIFO 管道的名称。关于更详尽的信息，用户可随时使用 help 命令查看帮助信息。

10.3.2 命名管道的创建

创建一个 FIFO 文件类似于创建一个普通文件，并且 FIFO 文件和其他文件一样，也是可以通过路径名来访问的。FIFO 管道通过函数 mkfifo 创建，函数原型如下：

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo (const char *pathname, mode_t mode);
```

返回：若成功则返回 0，若出错则返回-1。

该函数的第一个参数是一个普通的路径名，也就是创建后 FIFO 文件的名字。第二个参数与打开普通文件的 open 函数中的 mode 参数相同。如果 mkfifo 的第一个参数是一个已经存在的路径名时，则返回 EEXIST 错误，所以一般典型的调用代码首先会检查是否返回该错误，如果确实返回该错误，那么只要调用打开 FIFO 的函数就可以了。一般文件的 I/O 函数都可以用于 FIFO，如 close、read、write 等。

程序 10.4 演示了如何使用 mkfifo 函数来创建一个 FIFO。程序中从命令行参数中得到一个文件名，然后使用 mkfifo 函数创建 FIFO 文件。新创建的 FIFO 只具有读写权限。由于 FIFO 文件的特性，所以它被隐性地规定不具有执行权限。程序清单如 create_FIFO.c 所示。

【程序 10.4】创建一个命名管道：create_FIFO.c。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[ ])
{
    mode_t mode=0666;      /*新创建的 FIFO 模式*/
    if(argc != 2)
        {
        printf("USEMSG: create_FIFO { FIFO name}\n"); /*向用户提示程序使用帮助*/
        exit(1);
    }
    if((mkfifo(argv[1], mode))<0)    /*  使用 mkfifo 函数创建一个 FIFO 管道*/
        {
        perror("failed to mkfifo!\n");
        exit(1);
    }
    else
    {  /*输出 FIFO 文件的名称*/
        printf("you successfully create a FIFO name is : %s\n", argv[1]);
    }
return 0;
}
```

使用 gcc 编译 create_FIFO.c，并生成可执行文件 create_FIFO：

```
#gcc –o create_FIFO create_FIFO.c
```

运行程序，得到输出结果：

```
#./ create_FIFO
USEMSG: create_FIFO { FIFO name}
```

输入命令参数不对，打印出提示信息提示用户输入 FIFO 的文件名，下面输入正确的命令行参数：

```
#./ create_FIFO testFIFO
you successfully create a FIFO name is : testFIFO
```

再次运行程序，命令参数仍然为 testFIFO：

```
#./ create_FIFO testFIFO
mkfifo: File exists
```

程序使用 mkfifo 函数创建了一个 FIFO，名字是基于用户的输入文件名，可以看到，当要创建一个已经存在的 FIFO 时，程序会产生一个 EEXIST 的异常，相对应该异常，perror 函数打印了相应的帮助信息为 mkfifo: File exists。

另外，值得一提的是，命名管道 FIFO 比管道多了一个打开操作 open。原因很简单，命名管道是 Linux 下的一种文件类型，而管道不是。

然而，FIFO 的打开与其他文件的打开是不同的，FIFO 的打开规则是：

- 如果当前打开操作是为读而打开 FIFO 时，若已经有相应进程为写而打开该 FIFO，则当前打开操作将成功返回；否则，可能阻塞直到有相应进程为写而打开该 FIFO(当前打开操作设置了阻塞标志)；或者，成功返回(当前打开操作没有设置阻塞标志)。
- 如果当前打开操作是为写而打开 FIFO 时，若已经有相应进程为读而打开该 FIFO，则当前打开操作将成功返回；否则，可能阻塞直到有相应进程为读而打开该 FIFO(当前打开操作设置了阻塞标志)；或者，返回 ENXIO 错误(当前打开操作没有设置阻塞标志)。

10.3.3 命名管道的读写

使用命名管道的操作和使用普通文件十分相似，可以使用系统调用 open 打开一个命名管道，使用 read 和 write 函数对命名管道进行读写，使用 close 关闭一个命名管道，若要删除一个命名管道，则使用系统调用 unlink。这些函数均在第 6 章中进行了详细的介绍，下面重点介绍命名管道的读写。

从 FIFO 中读取数据的规则是：

- 如果一个进程为了从 FIFO 中读取数据而阻塞打开 FIFO，那么称该进程内的读操作为设置了阻塞标志的读操作。
- 如果有进程写打开 FIFO，且当前 FIFO 内没有数据，则对于设置了阻塞标志的读操作来说，将一直阻塞。对于没有设置阻塞标志读操作来说则返回-1，当前 errno 值为 EAGAIN，提醒以后再试。
- 对于设置了阻塞标志的读操作来说，造成阻塞的原因有两种：当前 FIFO 内有数据，但有其他进程在读这些数据；另外就是 FIFO 内没有数据。解阻塞的原因则是 FIFO 中有新的数据写入，不论新写入数据量的大小，也不论读操作请求多少数据量。

- 读打开的阻塞标志只对本进程第一个读操作施加作用，如果本进程内有多个读操作序列，则在第一个读操作被唤醒并完成读操作后，其他将要执行的读操作将不再阻塞，即使在执行读操作时，FIFO 中没有数据也一样(此时，读操作返回 0)。
- 如果没有进程写打开 FIFO，则设置了阻塞标志的读操作会阻塞。

注 意

如果 FIFO 中有数据，则设置了阻塞标志的读操作不会因为 FIFO 中的字节数小于请求读的字节数而阻塞，此时，读操作会返回 FIFO 中现有的数据量。

向 FIFO 中写入数据的规则是：

- 如果一个进程为了向 FIFO 中写入数据而阻塞打开 FIFO，那么称该进程内的写操作为设置了阻塞标志的写操作。
- 对于设置了阻塞标志的写操作，当要写入的数据量不大于 PIPE_BUF 时，Linux 将保证写入的原子性。如果此时管道空闲缓冲区不足以容纳要写入的字节数，则进入睡眠，直到当缓冲区中能够容纳要写入的字节数时，才开始进行一次性写操作。
- 当要写入的数据量大于 PIPE_BUF 时，Linux 将不再保证写入的原子性。FIFO 缓冲区一有空闲区域，写进程就会试图向管道写入数据，写操作在写完所有请求写的数据后返回。
- 对于没有设置阻塞标志的写操作，当要写入的数据量大于 PIPE_BUF 时，Linux 将不再保证写入的原子性。在写满所有 FIFO 空闲缓冲区后，写操作返回。
- 当要写入的数据量不大于 PIPE_BUF 时，Linux 将保证写入的原子性。如果当前 FIFO 空闲缓冲区能够容纳请求写入的字节数，写完后成功返回；如果当前 FIFO 空闲缓冲区不能够容纳请求写入的字节数，则返回 EAGAIN 错误，提醒以后再写。

说 明

原子操作(atomic operation)指的是由多步组成的操作。简单来讲，操作的原子性是指某一事务中的所有操作要么全部执行，要么全部不执行，不可能只执行所有步骤的一个子集。

程序 10.5 和程序 10.6 演示了使用 FIFO 来进行两个进程间通信的例子。在程序 write_fifo.c 中打开一个名为 fifo1 的 FIFO 文件，并分 10 次向这个 FIFO 中写入数据。在程序 read_fifo.c 中先打开 fifo1 文件，然后读取里面的数据并输出到标准输出。

【程序 10.5】 向 FIFO 写入数据：write_fifo.c。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <limits.h>
#define BUFES PIPE_BUF
int main(void)
```

```
{
    int fd ;
    int n, i ;
    char buf[BUFES];
    time_t tp;
    printf("I am %d\n",getpid());          /*说明进程的 ID*/
    if((fd=open("fifo1",O_WRONLY))<0)   /*以只写方式打开一个 FIFO，管道名为 fifo1*/
{
        printf("Open failed!\n");
        exit(1);
    }
    for ( i=0 ; i<10; i++)     /*循环 10 次向 FIFO 中写入数据*/
{
        time(&tp);              /*取系统当前时间*/
        n=sprintf(buf,"write_fifo %d sends %s",getpid(),ctime(&tp));
                            /*使用 sprintf  函数向 buf 中格式化写入进程 ID 和时间值*/
        printf("Send msg:%s\n",buf );
        if((write(fd, buf, n+1))<0)
 {    /*写入到 FIFO 中*/
            printf ("Write failed!\n");
            close(fd);                      /*关闭 FIFO 文件*/
            exit(1);
        }
        sleep(3);                          /*进程睡眠 3 秒，便于观察*/
    }
    close(fd);                             /*关闭 FIFO 文件*/
    exit(0);
}
```

程序中使用 open 函数打开一个名为 fifo1 的 FIFO 管道，并分 10 次向 fifo1 中写入字符串，其中的数据有当前进程 ID 及写入时的系统时间。并把这个数据串输出到标准输出，然后程序自动睡眠 3 秒。

说明

在 write_fifo.c 中，使用到的 ctime 函数是一个时间格式的转换函数，将系统当前时间转换成周、月、日、时分秒、年的格式。更详细的信息读者可以参考标准 C 函数库。

【程序 10.6】从 FIFO 读取数据：read_fifo.c。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <fcntl.h>
#include <unistd.h>
#define BUFES PIPE_BUF
int main(void)
{
```

```
        int fd;
        int len;
        char buf[BUFES];
        mode_t mode = 0666;                     /* FIFO 文件的权限 */
        if((fd=open("fifo1",O_RDONLY))<0)     /* 打开 FIFO 文件 */
    {
            Printf("Open failed!\n");
            exit(1);
        }
        while((len=read(fd,buf, BUFES))>0)     /*开始进行通信*/
            printf("Read_fifo read: %s",buf);
    close(fd);     /*关闭 FIFO 文件*/
        exit(0);
    }
```

程序中使用 open 函数以读方式打开一个名为 fifo1 的 FIFO 管道，并循环读出管道的数据，这里使用 while 循环的作用就是确保数据可以全部读出，因为在读 FIFO 管道数据时，默认的是一次性读取 PIPE_BUF 个字节，当管道中数据多于 PIPE_BUF 个字节时，一次性读出 PIPE_BUF-1 个字节，然后 read 函数返回，再打印数据到标准输出。

使用 gcc 分别编译上述两个程序如下：

```
#gcc –o write_fifo write_fifo.c
#gcc –o read_fifo read_fifo.c
```

在运行这两个程序之前，先要使用 mkfifo 命令创建一个名为 fifo1 的命名管道，命令如下：

```
#mkfifo –m 666 fifo1
```

此时，打开两个 Shell 分别运行程序 write_fifo 和程序 read_fifo。在一个 Shell 中输入如下命令，得到输出结果：

```
#./write_fifo
I am 2071
Send msg:write_fifo 2071 sends Sat Oct 17 09:20:00 2009
Send msg:write_fifo 2071 sends Sat Oct 17 09:20:03 2009
Send msg:write_fifo 2071 sends Sat Oct 17 09:20:06 2009
Send msg:write_fifo 2071 sends Sat Oct 17 09:20:09 2009
Send msg:write_fifo 2071 sends Sat Oct 17 09:20:12 2009
Send msg:write_fifo 2071 sends Sat Oct 17 09:20:15 2009
Send msg:write_fifo 2071 sends Sat Oct 17 09:20:18 2009
Send msg:write_fifo 2071 sends Sat Oct 17 09:20:21 2009
Send msg:write_fifo 2071 sends Sat Oct 17 09:20:24 2009
Send msg:write_fifo 2071 sends Sat Oct 17 09:20:27 2009
```

在另一个 Shell 中输入如下命令，得到输出结果：

```
#./read_fifo
read_fifo read: write_fifo 2071 sends Sat Oct 17 09:20:00 2009
read_fifo read: write_fifo 2071 sends Sat Oct 17 09:20:03 2009
read_fifo read: write_fifo 2071 sends Sat Oct 17 09:20:06 2009
read_fifo read: write_fifo 2071 sends Sat Oct 17 09:20:09 2009
```

```
read_fifo read: write_fifo 2071 sends Sat Oct 17 09:20:12 2009
read_fifo read: write_fifo 2071 sends Sat Oct 17 09:20:15 2009
read_fifo read: write_fifo 2071 sends Sat Oct 17 09:20:18 2009
read_fifo read: write_fifo 2071 sends Sat Oct 17 09:20:21 2009
read_fifo read: write_fifo 2071 sends Sat Oct 17 09:20:24 2009
read_fifo read: write_fifo 2071 sends Sat Oct 17 09:20:27 2009
```

上述例子可以扩展成客户端与服务器通信的实例，write_fifo 的作用类似于客户端，可以打开多个客户端向一个服务器发送请求信息，read_fifo 类似于服务器，它适时监控着 FIFO 的读出端，当有数据时，读出并进行处理，但是有一个关键的问题是，每一个客户端必须预先知道服务器提供的 FIFO 接口，如图 10.3 所示。

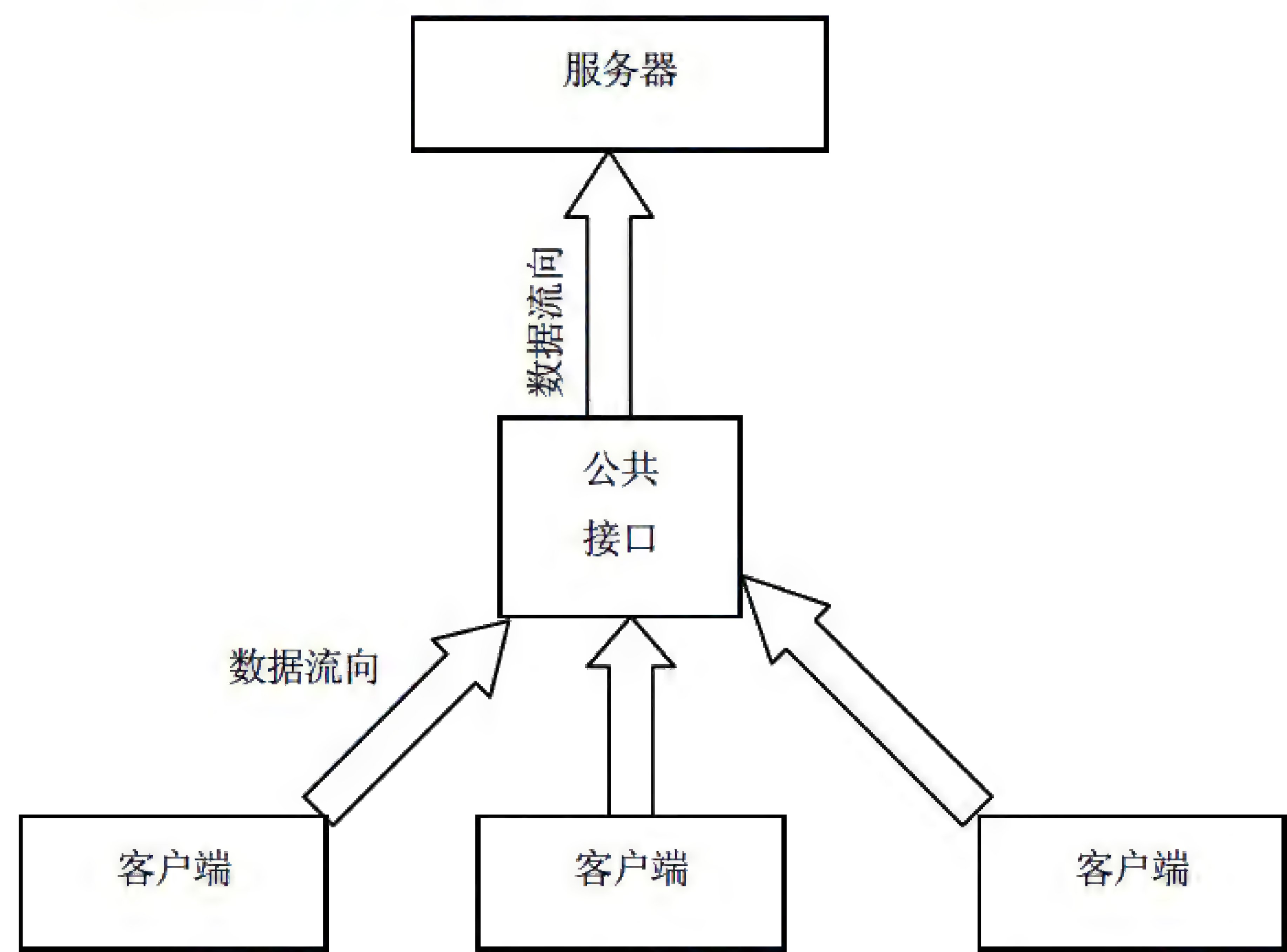


图 10.3　FIFO 在服务器与客户端通信应用的示意图

10.4 消息队列

消息队列是一种以链表式结构组织的一组数据，存放在内核中，是由各进程通过消息队列标识符来引用的一种数据传送方式。像其他两种 IPC 对象一样，也是由内核来维护。消息队列是 3 个 IPC 对象类型中最具有数据操作性的数据传送方式，在消息队列中可以随意根据特定的数据类型值来检索消息。

10.4.1　消息队列的概念

消息队列(也叫作报文队列)能够克服早期UNIX通信机制的一些缺点。比如作为早期UNIX通信机制之一的信号，它所能够传送的信息量是很有限的，后来虽然 POSIX 1003.1b 在信号的实时性方面作了拓展，使得信号在传递信息量方面有了相当程度的改进，但信号更像是一种“即时”的通信方式，它要求接收信号的进程在某个时间范围内对信号做出反应，因此该信号最多在

接收信号进程的生命周期内才有意义，信号所传递的信息是接近于随进程持续(process-persistent)的概念。管道及有名管道则是典型的随进程持续 IPC，并且，只能传送无格式的字节流无疑会给应用程序开发带来不便，另外，它的缓冲区大小也受到限制。

消息队列就是一个消息的链表。可以把消息看作一个记录，具有特定的格式及特定的优先级。对消息队列有写权限的进程可以向其中按照一定的规则添加新消息；对消息队列有读权限的进程则可以从消息队列中读取消息。消息队列是随内核持续(kernel-persistent)的。

说 明

- 随进程持续的定义为：IPC 一直存在，直至打开 IPC 对象的最后一个进程关闭该对象为止，如管道和有名管道。
- 随内核持续的定义为：IPC 一直持续到内核重新自举或者显示删除该对象为止。如消息队列、信号量及共享内存等。
- 随文件系统持续的定义为：IPC 一直持续到显示删除该对象为止。

目前主要有两种类型的消息队列：POSIX 消息队列及 System V 消息队列，System V 消息队列目前被大量使用。考虑到程序的可移植性，新开发的应用程序应尽量使用 POSIX 消息队列。

本书将主要介绍 System V 消息队列及其相应的 API。在没有声明的情况下，以下讨论中指的都是 System V 消息队列。

System V 消息队列是随内核持续的，只有在内核重起或者显示删除一个消息队列时，该消息队列才会真正被删除。因此系统中记录消息队列的数据结构(struct ipc_ids msg_ids)位于内核中，系统中的所有消息队列都可以在结构 msg_ids 中找到访问入口。

消息队列就是一个消息的链表。每个消息队列都有一个队列头，用结构 struct msg_queue 来描述。队列头中包含了该消息队列的大量信息，包括消息队列键值、用户 ID、组 ID、消息队列中消息数目等，甚至记录了最近对消息队列读写进程的 ID。用户可以访问这些信息，也可以设置其中的某些信息。

结构 msg_queue 用来描述消息队列头，存在于系统空间，定义如下：

```
struct msg_queue
{
    struct ipc_perm q_perm;
    time_t q_stime;            /* last msgsnd time */
    time_t q_rtime;            /* last msgrcv time */
    time_t q_ctime;            /* last change time */
    unsigned long q_cbytes;    /* current number of bytes on queue */
    unsigned long q_qnum;      /* number of messages in queue */
    unsigned long q_qbytes;    /* max number of bytes on queue */
    pid_t q_lspid;             /* pid of last msgsnd */
    pid_t q_lrpid;             /* last receive pid */
    struct list_head q_messages;
    struct list_head q_receivers;
```

```
        struct list_head q_senders;
};
```

结构 msqid_ds 用来设置或返回消息队列的信息，存在于用户空间，定义如下：

```
struct msqid_ds
{
        struct ipc_perm msg_perm;
        struct msg *msg_first;          /* first message on queue,unused    */
        struct msg *msg_last;           /* last message in queue,unused */
        time_t msg_stime;   /* last msgsnd time */
        time_t msg_rtime;   /* last msgrcv time */
        time_t msg_ctime;   /* last change time */
        unsigned long   msg_lcbytes; /* Reuse junk fields for 32 bit */
        unsigned long   msg_lqbytes; /* ditto */
        unsigned short msg_cbytes;   /* current number of bytes on queue */
        unsigned short msg_qnum;      /* number of messages in queue */
        unsigned short msg_qbytes;   /* max number of bytes on queue */
        pid_t msg_lspid;      /* pid of last msgsnd */
        pid_t msg_lrpid;      /* last receive pid */
};
```

图 10.4 说明了内核与消息队列是怎样建立起联系的。其中 struct ipc_ids msg_ids 是内核中记录消息队列的全局数据结构；struct msg_queue 是每个消息队列的队列头。

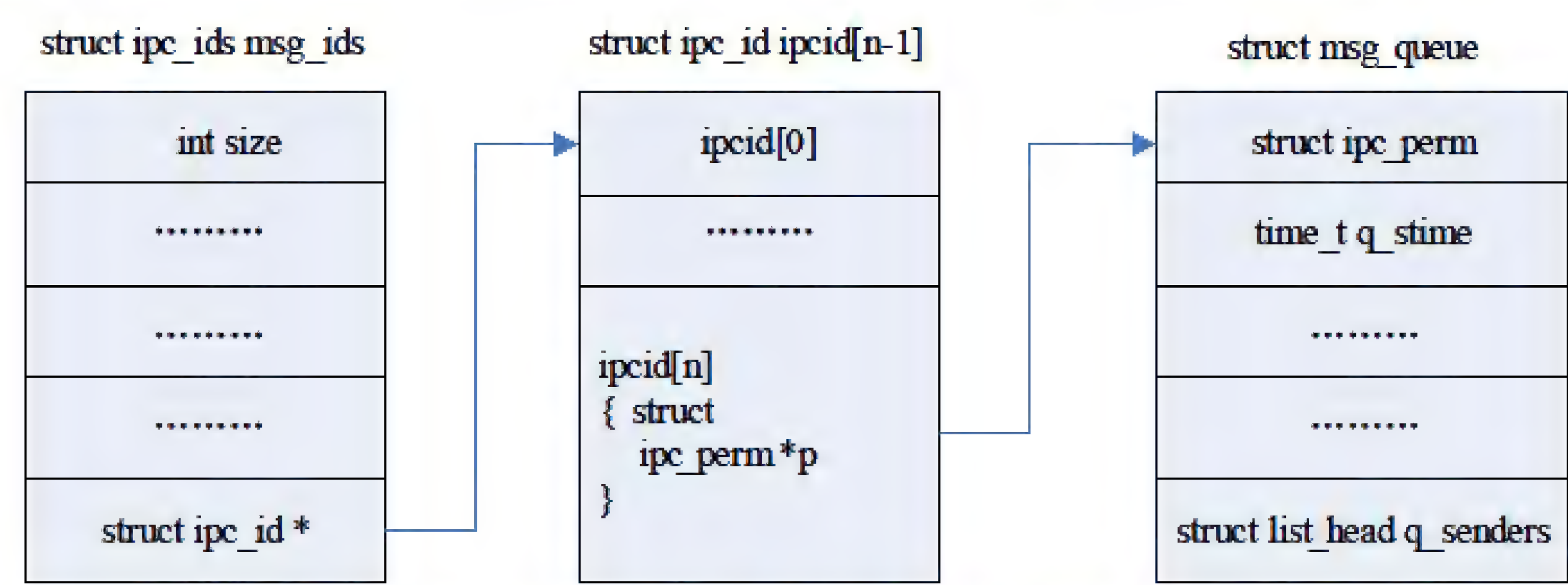


图 10.4　内核数据结构与消息队列

从上图可以看出，全局数据结构 struct ipc_ids msg_ids 可以访问每个消息队列头的第一个成员 struct ipc_perm；而每个 struct ipc_perm 能够与具体的消息队列对应起来是因为在该结构中，有一个 key_t 类型成员 key，而 key 则唯一确定一个消息队列。ipc_perm 结构定义如下：

```
struct ipc_perm
{   /*内核中记录消息队列的全局数据结构 msg_ids 能够访问到该结构*/
key_t     key;      /*该键值唯一对应一个消息队列*/
uid_t     uid;      /*所有者的有效用户 ID*/
gid_t     gid;      /*所有者的有效组 ID */
uid_t     cuid;     /*创建者的有效用户 ID */
gid_t     cgid;     /*创建者的有效组 ID */
mode_t    mode;     /*此对象的访问权限*/
unsigned long seq; /*对象的序号*/
};
```

10.4.2 消息队列的创建与打开

消息队列的内核持续性要求每个消息队列都在系统范围内对应唯一的键值，所以，要获得一个消息队列的引用标识符(ID)——即创建或打开消息队列，只需提供该消息队列的键值即可。

说 明

消息队列的引用标识符也可以称为消息队列的 ID 号，就像进程 ID 一样，用来唯一标识系统中的某一进程。消息队列的 ID 是由在系统范围内唯一的键值生成的，而键值可以看作对应系统内的一条路径。

获得特定文件名的键值的系统调用是 ftok，函数原型如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok (char*pathname, char proj);
```

返回：若成功则返回消息队列的一个键值，若失败则返回-1。

ftok 返回与路径 pathname 相对应的一个键值。该函数并不直接对消息队列操作，但在调用 msgget()来获得消息队列的标识符前，往往要调用该函数。

创建或打开一个消息队列的系统调用为 msgget，函数原型如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget (key_t key, int msgflg);
```

返回：若成功则为消息队列的引用标识符(ID)，若失败则返回-1。

参数 key 是一个键值，由 ftok 获得；msgflg 参数是一些标志位，可以取以下值：IPC_CREAT、IPC_EXCL、IPC_NOWAIT 或三者的逻辑或结果。该调用返回与键值 key 相对应的消息队列标识符。

在以下两种情况下，该调用将创建一个新的消息队列：

- 如果没有消息队列与键值 key 相对应，并且 msgflg 中包含了 IPC_CREAT 标志位。
- key 参数为 IPC_PRIVATE。

提 示

参数 key 设置成常数 IPC_PRIVATE 并不意味着其他进程不能访问该消息队列，只意味着即将创建新的消息队列。

另外，当调用 msgget 函数创建一个消息队列时，它相应的 msqid_ds 结构被初始化。ipc_perm 结构中的各成员被设置为相应值，msg_qnum、msg_lspid、msg_lrpid、msg_stime、msg_rtime 都被设置为 0，msg_qbytes 被设置为系统限制值，msg_ctime 被设置为当期时间(读者将在本节结束时的例子中看到)。

10.4.3 消息队列的读写

使用消息队列进行进程间的通信，就是要对消息队列进行读和写的操作。写操作即向消息

队列中发送数据，而读操作则是从消息队列中接收(读取)数据。

消息队列所传递的消息由两部分组成，即消息的类型和所传递的数据。一般用数据结构 struct msgbuf 来表示，通常消息类型是一个正的长整数，而数据则根据需要设定。设定一个传递 1024 字节长度的消息可将结构定义如下：

```
struct msgbuf
{
long msgtype;
char msgtext[1024];
};
```

msgtype 成员代表消息类型，从消息队列中读取消息的一个重要依据就是消息的类型；msgtext 是消息内容，当然长度不一定为 1024。因此，对于发送消息来说，首先预置一个 msgbuf 缓冲区并写入消息类型和内容，调用相应的发送函数即可；对读取消息来说，首先分配这样一个 msgbuf 缓冲区，然后把消息读入该缓冲区即可。

1. 向消息队列发送数据

向消息队列发送数据系统调用的函数原型如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd (int msqid, const void *prt, size_t nbytes, int flags);
```

返回：若成功则返回 0，若出错则返回-1。

msgsnd 函数的作用是向一个消息队列发送一个消息，该消息被添加到队列的末尾。参数 msqid 代表消息队列的引用标识符(ID)，参数 prt 是一个 void 类型指针，指向要发送的消息，参数 nbytes 是以字节表示的消息的数据长度，参数 flags 用于指定消息队列满时的处理方法。

对发送消息来说，有意义的 flags 标志为 IPC_NOWAIT，指明在消息队列没有足够空间容纳要发送的消息时，msgsnd 是否等待。当消息队列满时，如果设置了 IPC_NOWAIT 位，则立刻出错返回，否则发送消息的进程被阻塞，直至消息队列中有空间或消息队列被删除时，函数返回。

造成 msgsnd 等待的条件有两种：

- 当前消息的大小与当前消息队列中的字节数之和超过了消息队列的总容量。
- 当前消息队列中的消息数(单位“个”)不小于消息队列的总容量(单位“字节数”)，此时，虽然消息队列中的消息数目很多，但基本上都只有一个字节。

msgsnd 解除阻塞的条件有 3 个：

- 不满足上述两个条件，即消息队列中有容纳该消息的空间。
- msqid 代表的消息队列被删除。
- 调用 msgsnd 的进程被某个信号中断。

2. 从消息队列接收数据

从消息队列接收数据系统调用的函数原型如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv (int msqid, void *prt, size_t nbytes, long type, int flags);
```

返回：若成功则返回消息的数据长度，若出错则返回-1。

此函数用于从指定的消息队列中读取一个消息数据。参数 msqid 代表消息队列的引用标识符，prt 是一个 void 类型指针，指向存放消息的缓冲区，nbytes 是以字节表示的要接收的消息的数据长度，flags 用于指定消息队列满时的处理方法，其取值为表 10.1 中所示的常值或几个常值的逻辑或。

表 10.1　flags 的取值及其含义

flags 取值	含　义
IPC_NOWAIT	如果没有满足条件的消息，调用立即返回，此时 errno=ENOMSG
IPC_EXCEPT	与 type>0 配合使用，返回队列中第一个类型不为 type 的消息
MSG_NOERROR	如果队列中满足条件的消息内容大于所请求的 nbytes 字节，则把该消息截断，截断部分将丢失

参数 type 用于表示要接收的消息的类型，其值如表 10.2 所示。

表 10.2　type 的取值及其含义

type 取值	含　义
type=0	接收消息队列中的第一条消息
type>0	接收消息队列中类型值等于 type 的第一条消息
type<0	接收消息队列中类型值小于等于 type 的绝对值的所有消息中类型值最小的那一条消息

msgrcv 解除阻塞的条件有 3 个：

- 消息队列中有了满足条件的消息。
- msqid 代表的消息队列被删除。
- 调用 msgrcv 的进程被某个信号中断。

10.4.4　获得或设置消息队列属性

消息队列的信息基本上都保存在消息队列头中，因此，可以分配一个类似于消息队列头的结构 struct msqid_ds 来返回消息队列的属性；同样可以设置该数据结构。关于消息队列属性的操作的系统调用如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl (int msqid, int cmd, struct msqid_ds *buf);
```

返回：若成功则返回 0，否则返回-1。

该系统调用对由 msqid 标识的消息队列执行 cmd 操作，共有 3 种 cmd 操作：IPC_STAT、IPC_SET、IPC_RMID，它们的含义分别如下：

- IPC_STAT：该命令用来获取消息队列信息，返回的信息存贮在 buf 指向的 msqid_ds 结构中。
- IPC_SET：该命令用来设置消息队列的属性，要设置的属性存储在 buf 指向的 msqid_ds 结构中；可设置属性包括 msg_perm.uid、msg_perm.gid、msg_perm.mode 及 msg_qbytes，同时，也影响 msg_ctime 成员。
- IPC_RMID：删除 msqid 标识的消息队列。

消息队列应用相对较简单，例子程序 10.7 基本上覆盖了对消息队列的所有操作，同时，程序输出结果有助于加深对前面所讲的某些规则及消息队列限制的理解。

程序 10.7 首先创建一个新的消息队列后，第 1 次调用 msg_stat 子函数打印出该消息队列的相关信息(默认属性)；然后向消息队列发送一个消息(字符数据“a”)后，第 2 次调用 msg_stat 打印消息队列此时的相关信息；接着再调用 msgrcv 从消息队列中读取消息后，第 3 次调用 msg_stat 打印消息队列此时的相关信息；最后试图使用 root 权限(注意运行此程序时请使用 root 用户)更改消息队列此时的相关属性，并再次调用 msg_stat 打印更改后的消息队列的信息。程序的最后，调用 msgctl 关闭该消息队列。

【程序 10.7】消息队列的使用举例：msg_app.c。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>
void msg_stat(int,struct msqid_ds );

int main(void)
{
int gflags,sflags,rflags;
key_t key;
int msgid;
int reval;
struct msgsbuf
{
        int mtype;
        char mtext[1];
    }msg_sbuf;          /*发送消息缓冲区数据结构*/
struct msgmbuf
    {
        int mtype;
        char mtext[10];
    }msg_rbuf;          /*接收消息缓冲区数据结构*/
struct msqid_ds msg_ginfo,msg_sinfo;
char* msgpath="/unix/msgqueue";
key=ftok(msgpath,'a');   /*获取消息队列键值*/
gflags=IPC_CREAT|IPC_EXCL;
```

```
msgid=msgget(key,gflags|00666);  /*调用 msgget 创建消息队列*/
if(msgid==-1)
{
        printf("msg create error\n");
        return;
}
msg_stat(msgid,msg_ginfo);
/*创建一个消息队列后，输出消息队列默认属性。第 1 次调用 msg_stat 子函数*/
sflags=IPC_NOWAIT;    /*消息队列满时，msgsnd 不等待，立刻出错返回*/
msg_sbuf.mtype=10;
msg_sbuf.mtext[0]='a';    /*将要发送的消息数据*/
reval=msgsnd(msgid,&msg_sbuf,sizeof(msg_sbuf.mtext),sflags); /*调用 msgsnd 发送消息*/
if(reval==-1)
{
        printf("message send error\n");
}
msg_stat(msgid,msg_ginfo);
/*成功发送一个消息后，输出此时消息队列属性。第 2 次调用 msg_stat  子函数*/
rflags=IPC_NOWAIT|MSG_NOERROR;          /*含义见表 10.1*/
reval=msgrcv(msgid,&msg_rbuf,4,10,rflags);
/*调用 msgrcv 接收消息，接收数据长度为 4，type＞0，含义见表 10.2*/
if(reval==-1)
{
        printf("read msg error\n");
    }
else
{
        printf("read from msg queue %d bytes\n",reval);    /*打印接收到数据的字节数*/
    }
msg_stat(msgid,msg_ginfo);
/*从消息队列中读出消息后，再次输出消息队列属性。第 3 次调用 msg_stat 子函数*/
msg_sinfo.msg_perm.uid=8;
/*试图更改消息队列的默认属性(要求 root 用户权限)，所有者有效用户 ID 更改为 8*/
msg_sinfo.msg_perm.gid=8;
/*消息队列的所有者有效组 ID 更改为 8*/
msg_sinfo.msg_qbytes=16388;
/*消息队列可容纳最大字节数更改为 16388(默认为 16384)*/
reval=msgctl(msgid,IPC_SET,&msg_sinfo);    /*调用 msgctl 设置消息队列属性*/
if(reval==-1)
{
        printf("msg set info error\n");
        return;
}
msg_stat(msgid,msg_ginfo);  /*验证设置消息队列属性。第 3 次调用 msg_stat  子函数*/
reval=msgctl(msgid,IPC_RMID,NULL);  /* 操作完毕，调用 msgctl 删除消息队列*/
if(reval==-1)
{
        printf("unlink msg queue error\n");
        return;
}
```

```
return 0;
}

void msg_stat (int msgid,struct msqid_ds msg_info)
{
int reval;
sleep(1);    /*只是为了后面输出时间的方便*/
reval=msgctl(msgid,IPC_STAT,&msg_info);   /*调用 msgctl 获得消息队列属性信息*/
if(reval==-1)
{
        printf("get msg info error\n");
        return;
}
printf("\n");
printf("current number of bytes on queue is %d\n",msg_info.msg_cbytes);
printf("number of messages in queue is %d\n",msg_info.msg_qnum);
printf("max number of bytes on queue is %d\n",msg_info.msg_qbytes);
/*每个消息队列的容量(字节数)都有限制 MSGMNB，值的大小因系统而异。在创建*/
/*新的消息队列时，msg_qbytes 的默认值就是 MSGMNB*/
printf("pid of last msgsnd is %d\n",msg_info.msg_lspid);
/*最近一个执行 msgsnd 函数的进程 ID*/
printf("pid of last msgrcv is %d\n",msg_info.msg_lrpid);
/*最近一个执行 msgrcv 函数的进程 ID*/
printf("last msgsnd time is %s", ctime(&(msg_info.msg_stime)));
/*最近一次执行 msgsnd 函数的时间。ctime()将时间转变成周、月、日、时分秒、年的*/
/*形式，属于标准 C 函数*/
printf("last msgrcv time is %s", ctime(&(msg_info.msg_rtime)));
/*最近一次执行 msgrcv 函数的时间*/
printf("last change time is %s", ctime(&(msg_info.msg_ctime)));
/*最近一次改变该消息队列的时间*/
printf("msg uid is %d\n",msg_info.msg_perm.uid);/*消息队列所有者的有效用户 ID*/
printf("msg gid is %d\n",msg_info.msg_perm.gid); /*消息队列所有者的有效组 ID*/
}
```

使用 gcc 编译 msg_app.c，并生成可执行文件 msg_app：

```
#gcc –o msg_app msg_app.c
```

运行程序，得到输出结果：

```
#./ msg_app
current number of bytes on queue is 0
number of messages in queue is 0
max number of bytes on queue is 16384   /*每个消息队列最大容量(字节数)*/
pid of last msgsnd is 0
pid of last msgrcv is 0
last msgsnd time is Thu Jan   1 08:00:00 1970
last msgrcv time is Thu Jan   1 08:00:00 1970
last change time is Wed Oct 14 22:57:28 2009
msg uid is 0
msg gid is 0
```

```
current number of bytes on queue is 1
number of messages in queue is 1
max number of bytes on queue is 16384
pid of last msgsnd is 2562
pid of last msgrcv is 0
last msgsnd time is Wed Oct 14 22:57:29 2009
last msgrcv time is Thu Jan   1 08:00:00 1970
last change time is Wed Oct 14 22:57:28 2009
msg uid is 0
msg gid is 0
read from msg queue 1 bytes

current number of bytes on queue is 0
number of messages in queue is 0
max number of bytes on queue is 16384
pid of last msgsnd is 2562
pid of last msgrcv is 2562
last msgsnd time is Wed Oct 14 22:57:29 2009
last msgrcv time is Wed Oct 14 22:57:30 2009
last change time is Wed Oct 14 22:57:28 2009
msg uid is 0
msg gid is 0

current number of bytes on queue is 0
number of messages in queue is 0
max number of bytes on queue is 16388   /*超级用户修改了消息队列的最大容量*/
pid of last msgsnd is 2562
pid of last msgrcv is 2562          /*对操作消息队列进程的跟踪*/
last msgsnd time is Wed Oct 14 22:57:29 2009
last msgrcv time is Wed Oct 14 22:57:30 2009
last change time is Wed Oct 14 22:57:31 2009   /*msgctl()调用对 msg_ctime 有影响*/
msg uid is 8
msg gid is 8
```

读者可尝试自行分析程序的输出结果，笔者已经在程序的注释中给出了详细的解释，这里不再赘述。

另外值得一提的是，消息队列的限制性，即每个消息队列的容量(所能容纳的字节数)都是有一定的限制的，该值因系统不同而不同。在上面的实例程序 10.7 中，输出了 Ubuntu 12.04 的限制。

另一个限制是每个消息队列所能容纳的最大消息数，在 Red Hat 9.0 中，该限制是受消息队列容量制约的，消息个数要小于消息队列的容量(字节数)。

说 明

上述两个限制是针对每个消息队列而言的，系统对消息队列的限制还有系统范围内的最大消息队列个数，以及整个系统范围内的最大消息数。一般来说，实际开发过程中不会超过这个限制。

10.5 共享内存

共享内存可以说是 Linux 下最快速、最有效的进程间通信方式。两个不同进程 A、B 共享内存的意思是，同一块物理内存被映射到进程 A、B 各自的进程地址空间，进程 A 可以即时看到进程 B 对共享内存中数据的更新，反之，进程 B 也可以即时看到进程 A 对共享内存中数据的更新。

10.5.1 共享内存的概念

共享内存从字面意义解释就是多个进程可以把一段内存映射到自己的进程空间，以此来实现数据的共享及传输，这也是所有进程间通信方式中最快的一种，共享内存是存在于内核级别的一种资源。

在 Shell 中可以使用 ipcs 命令来查看当前系统 IPC 中的状态，在文件系统中/proc 目录下有对其描述的相应文件。下列是笔者在当前系统中使用 ipcs 命令后的输出结果：

```
# ipcs
------ Shared Memory Segments --------
key         shmid       owner       perms       bytes       nattch      status
0x00000001 32768        root        600         655360      2
0xffffffff 589825       root        0           4096        0

------ Semaphore Arrays --------
key         semid       owner       perms       nsems

------ Message Queues --------
key         msqid       owner       perms       used-bytes  messages
```

在系统内核为一个进程分配内存地址时，通过分页机制可以让一个进程的物理地址不连续，同时也可以让一段内存同时分配给不同的进程。共享内存机制就是通过该原理来实现的，共享内存机制只是提供数据的传送，如何控制服务器端和客户端的读写操作互斥，这就需要一些其他的辅助工具，例如信号量的概念。

采用共享内存通信的一个显而易见的好处是效率高，因为进程可以直接读写内存，而不需要任何数据的复制。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据复制，而共享内存则只复制两次数据：一次从输入文件到共享内存区，另一次从共享内存区到输出文件。实际上，进程之间在共享内存时，并不总是读写少量数据后就解除映射，有新的通信时，再重新建立共享内存区域。而是保持共享区域，直到通信完毕为止，这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存中的内容往往是在解除映射时才写回文件的。因此，采用共享内存的通信方式效率是非常高的。

共享内存的最大不足之处在于，由于多个进程对同一块内存区域具有访问的权限，各个进程之间的同步问题显得尤为重要。必须控制同一时刻只有一个进程对共享内存区域写入数据，否则将造成数据的混乱。同步控制问题可通过下一节介绍的信号量来解决。

对于每一个共享存储段，内核会为其维护一个 shmid_ds 类型的结构体(shmid_ds 结构体定义在头文件<sys/shm.h>中)。shmid_ds 结构体定义如下：

```
struct shmid_ds
{
    struct ipc_perm shm_perm;        /*对应于该共享内存的 ipc_perm 结构*/
    size_t   shm_segsz;              /*以字节表示的共享内存区域的大小*/
    pid_t   shm_lpid;                /*最近一次调用 shmop 函数的进行 ID*/
    pid_t   shm_cpid;                /*创建该共享内存的进程 ID*/
    unsigned short shm_lkcnt;        /*共享内存区域被锁定的时间数*/
    unsigned long shm_nattch;        /*当期使用该共享内存的进程数*/
    time_t   shm_atime;              /*最近一次附加操作的时间*/
    time_t   shm_dtime;              /*最近一次分离操作的时间*/
    time_t   shm_ctime;              /*最近一次修改的时间*/
};
```

说 明

结构体 shmid_ds 会根据不同的系统内核版本而略有不同，并且在不同的系统中会对共享存储段的大小有限制，在应用时请查询相应的系统手册。

下面详细介绍有关共享内存的函数调用。

10.5.2 共享内存的相关操作

对于 System V 共享内存，主要的几个 API 有 shmget、shmat、shmdt 及 shmctl。下面分别对它们进行介绍。

1. 创建或打开共享内存

要使用共享内存，首先要创建一个共享内存区域，创建共享内存的函数调用如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget (key_t key, int size, int flag);
```

返回：若成功则返回共享内存 ID，若出错则返回-1。

shmget 函数除了可以用于创建一个新的共享内存外，也可用于打开一个已存在的共享内存。其中，参数 key 表示所要创建或打开的共享内存的键值。size 表示共享内存区域的大小，只在创建一个新的共享内存时生效。参数 flag 表示调用函数的操作类型，也可用于设置共享内存的访问权限，两者通过逻辑或表示。参数 key 和 flag 决定了调用函数 shmget 的作用，相应的约定如下：

- 当 key 为 IPC_PRIVATE 时，创建一个新的共享内存，此时参数 flag 的取值无效。
- 当 key 不为 IPC_PRIVATE 时，且 flag 设置了 IPC_CREAT 位，而没有设置 IPC_EXCL 位，则执行操作由 key 取值决定。如果 key 为内核中某个已存在的共享内存的键值，则执行打开这个键的操作；反之，则执行创建共享内存的操作。

➢ 当 key 不为 IPC_PRIVATE 时，且 flag 设置了 IPC_CREAT 位和 IPC_EXCL 位，则只执行创建共享内存的操作。参数 key 的取值应与内核中已存在的任何共享内存的键值都不相同，否则函数调用失败，返回 EEXIST 错误，所以一般典型的调用代码首先会检查是否返回该错误，如果确实返回该错误，那么只要调用打开共享内存的函数就可以了(即将 flag 设置为 IPC_CREAT，而不设置 IPC_EXCL)。

另外，当调用 shmget 函数创建一个新的共享内存时，此共享内存的 shmid_ds 结构被初始化。ipc_ perm 中的各个域被设置为相应的值，shm_lpid、shm_nattch、shm_atime、shm_dtime 被初始化为 0，shm_ctime 被设置为系统当期时间。

程序 10.8 演示了使用 shmget 函数创建一块共享内存。程序中在调用 shmget 函数时指定 key 参数值为 IPC_PRIVATE，这个参数的意义是创建一个新的共享内存区，当创建成功后使用 shell 命令 ipcs -m 来显示目前系统下共享内存的状态。代码清单如 create_shm.c 所示。

说明

ipcs 命令表示显示当前系统下进程间通信方式的种类及状态信息，-m 选项表示共享内存。

【程序 10.8】创建共享内存：create_shm.c。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdlib.h>
#include <stdio.h>
#define BUFSZ 4096
int main (void)
{
    int shm_id;        /*共享内存标识符*/
    shm_id=shmget (IPC_PRIVATE, BUFSZ, 0666 );    /*创建共享内存*/
    if (shm_id<0)
{
        printf("shmget failed!\n");
        exit (1);     /* shmget  出错退出*/
    }
    printf ("creat a shared memory segment successfully: %d \n", shm_id);
    system( "ipcs -m");         /*调用 ipcs 命令查看 IPC*/
    exit(0);
}
```

使用 gcc 编译 create_shm.c，并生成可执行文件 create_shm：

```
#gcc –o create_shm create_shm.c
```

运行程序，得到输出结果：

```
#./ create_shm
creat a shared memory segment successfully: 655362

------ Shared Memory Segments --------
key          shmid        owner        perms        bytes        nattch       status
```

```
0x00000001 32768     root     600     655360     2
0xffffffff 589825   root     0     4096     0
0x00000000 655362     root     666     4096     0
```

上述程序中使用 shmget 函数来创建一段共享内存，并在结束前调用了系统 shell 命令 ipcs –m 来查看当前系统 IPC 状态。对比 10.5.1 小节开始时 ipcs 命令的输出结果，可以发现系统的确多了一个共享内存段，且该共享内存 ID 为 655362。

2. 附加

当一个共享内存创建或打开后，某个进程如果要使用该共享内存则必须将此内存区域附加到它的地址空间，附加操作的系统调用如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
void *shmat ( int shmid, const void *addr, int flag );
```

返回：若成功则返回指向共享内存段的指针，若出错则返回-1。

参数 shmid 指定要引入的共享内存，参数 addr 与 flag 组合说明要引入的地址值，通常只有两种用法，addr 为 0(即指向 NULL)，表明让内核来决定第 1 个可以引入的位置。addr 非零，并且 flag 中指定 SHM_RND，则此段引入到 addr 所指向的位置(此操作不推荐使用，因为不会只对一种硬件上运行应用程序，为了程序的通用性推荐使用第 1 种方法)，在 flag 参数中可以指定要引入的方式(读写方式指定)。

shmat 函数成功执行后，会将 shmid 所表示共享内存段的 shmid_ds 结构的 shm_nattch 计数器的值加 1。

3. 分离

当进程对共享内存段的操作完成后，应调用 shmdt 函数，作用是将指定的共享内存段从当前进程空间中脱离出去。函数原型如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmdt (void *addr);
```

返回：若成功则返回 0，否则返回-1。

此函数仅用于将共享内存区域与进程的地址空间分离，并不删除共享内存本身。参数 addr 是调用 shmat 函数时的返回值。shmdt 函数成功执行后，将该共享内存的 shmid_ds 结构中的 shm_nattch 计数器减 1。

4. 共享内存的控制

由于共享内存这一特殊的资源类型，使它不同于普通的文件，因此，系统需要为其提供专有的操作函数，而这无疑增加了程序员开发的难度(需要记忆额外的专有函数)。使用函数 shmctl 可以对共享内存段进行多种操作，其函数原型如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
```

```
#include <sys/shm.h>
int shmctl (int shmid, int cmd, struct shmid_ds *buf );
```

返回：若成功则返回 0，否则返回-1。

参数 shmid 为所要操作的共享内存段的标识符，struct shmid_ds 型指针参数 buf 的作用与参数 cmd 的值相关，参数 cmd 指明了所要进行的操作，其解释如表 10.3 所示。

表 10.3　cmd 的取值及其含义

cmd 取值	含　义
IPC_STAT	取 shmid 所指向内存共享段的 shmid_ds 结构，对参数 buf 指向的结构赋值
IPC_SET	使用 buf 指向的结构对 sh_mid 段的相关结构赋值，只对以下几个域有作用，shm_perm.uid shm_perm.gid 及 shm_perm.mode
IPC_RMID	删除 shmid 所指向的共享内存段，只有当 shmid_ds 结构的 shm_nattch 域为零时，才会真正执行删除命令，否则不会删除该段。注意此命令的请求规则与 IPC_SET 命令相同
SHM_LOCK	锁定共享内存段在内存，此命令只能由超级用户请求
SHM_UNLOCK	对共享内存段解锁，此命令只能由超级用户请求

说 明

对于上表中的 IPC_SET 参数选项，只有具备以下条件的进程才可以请求：(1) 进程的用户 ID 等于 shm_perm.cuid 或者等于 shm_perm.uid；(2) 超级用户特权进程。

程序 10.9 和程序 10.10 是两个进程通过共享内存进行通信的范例，这两个程序基本涵盖了共享内存的所有操作的系统调用。在程序 10.8 中，我们成功创建了一个 ID 号为 655362 的共享内存段，在下面的这两个程序中，将使用这段共享内存进行两个进程间的通信，共享内存 ID(程序中为 int 型变量 shm_id)以命令行参数的形式传递给进程。

程序 10.9 的功能是写共享内存，即分 10 次向共享内存中写入 people 结构体的成员数据(姓名和年龄)，数据的值由 for 循环自动生成。程序的最后使用 shmdt 将共享内存段从当前的进程空间中脱离掉。代码清单如 write_shm.c 所示。

【程序 10.9】向共享内存中写入数据：write_shm.c。

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>

typedef struct
{
        char name[4];
        int age;
} people;

int main(int argc, char** argv)
{
```

```
int shm_id,i;
        char temp;
        people *p_map;
if ( argc != 2 )    /*命令行参数错误*/
          {
           printf ("USAGE: atshm <identifier>" );      /*打印帮助消息*/
           exit (1);
      }
     shm_id = atoi(argv[1]);                        /*得到要引入的共享内存段*/
        p_map=(people *)shmat(shm_id,NULL,0);
temp='a';
        for(i = 0;i<10;i++)
          {
             temp+=1;
             memcpy((*(p_map+i)).name,&temp,1);
             (*(p_map+i)).age=20+i;
          }
        if(shmdt(p_map)==-1)
     {
            perror("detach error!\n");
     }
return 0;
}
```

程序 10.10 的功能是读取共享内存，即分 10 次从共享内存中读出 people 结构体的成员数据。程序的最后同样使用 shmdt 将共享内存段从当前的进程空间中脱离掉。代码清单如 read_shm.c 所示。

【程序 10.10】从共享内存读取数据：read_shm.c。

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>

typedef struct
{
        char name[4];
        int age;
} people;

int main(int argc, char** argv)
{
        int shm_id,i;
        people *p_map;
        if (argc != 2)        /*命令行参数错误*/
          {
          printf ( "USAGE: atshm <identifier>" );      /*打印帮助消息*/
          exit (1 );
      }
shm_id = atoi(argv[1]);    /*得到要引入的共享内存段*/
```

```
        p_map = (people*)shmat(shm_id,NULL,0);
        for(i = 0;i<10;i++)
          {
              printf( "name:%s      ",(*(p_map+i)).name );
printf( "age %d\n",(*(p_map+i)).age );
          }
        if(shmdt(p_map)==-1)
    {
              perror("detach error!\n");
    }
        return 0;
}
```

使用 gcc 分别编译这两个程序，并生产各自的可执行文件：

```
#gcc –o write_shm write_shm.c
#gcc –o read_shm read_shm.c
```

分别运行程序，得到输出结果：

```
#./ write_shm 655362          /*注意输入命令行参数——共享内存 ID*/
#./ read_shm 655362           /*注意输入命令行参数——共享内存 ID*/
name:b     age 20
name:c     age 21
name:d     age 22
name:e     age 23
name:f     age 24
name:g     age 25
name:h     age 26
name:i     age 27
name:j     age 28
name:k     age 29
```

从中可以看到，程序得到了正确的输出结果。read_shm 成功读取了之前由 write_shm 向共享内存 655362 中写入的数据。

10.6 信号量

信号量的原理是一种数据操作锁的概念，它本身不具备数据交换的功能，而是通过控制其他的通信资源(如文件、外部设备等)来实现进程间通信。信号量本身不具备数据传输的功能，它只是一种外部资源的标识。本节将深入介绍信号量的操作。

10.6.1 信号量的概念

信号量(Semaphore)，有时也被称为信号灯，是在多进程环境下使用的一种设施，它负责协调各个进程，以保证它们能够正确、合理地使用公共资源。信号量分为单值和多值两种，前者只能被一个进程获得，后者可以被若干个进程获得。

以一个停车场为例。简单起见，假设停车场只有 3 个车位，一开始 3 个车位都是空的。这时如果同时来了 5 辆车，看门人允许其中 3 辆直接进入，然后放下拦车杆，剩下的车则必须在入口等待，在此后来的车也都不得不在入口处等待。这时，有一辆车离开停车场，看门人得知后，打开拦车杆，放入外面的一辆进去，如果又离开两辆，则又可以放入两辆，如此往复。

在这个停车场系统中，车位是公共资源，每辆车好比一个进程，看门人所起的就是信号量的作用。

信号量，是可以用来保证两个或多个关键代码段不被并发调用。在进入一个关键代码段之前，进程必须获取一个信号量；一旦该关键代码段完成了，那么该进程必须释放信号量。其他想进入该关键代码段的进程必须等待直到第一个进程释放信号量。

抽象地来讲，信号量的特性如下：信号量是一个非负整数(车位数)，所有通过它的进程/线程(车辆)都会将该整数减 1(通过它当然是为了使用公共资源)，当该整数值为零时，所有试图通过它的进程都将处于等待状态。在信号量上我们定义两种操作：Wait(等待)和 Release(释放)。当一个进程调用 Wait 操作时，它要么得到资源然后将信号量减 1，要么一直等下去(指放入阻塞队列)，直到信号量大于等于 1 时。Release(释放)实际上是在信号量上执行加 1 操作，对应于车辆离开停车场，该操作之所以叫作“释放”是因为释放了由信号量守护的资源。

事实上，在信号量的实际应用中，是不能单独定义一个信号量的，而只能定义一个信号量集，其中包含一组信号量，同一信号量集中的信号量使用同一个引用 ID，这样的设置是为了多个资源或同步操作的需要。每个信号量集都有一个与之对应结构，其中记录了信号量集的各种信息，该结构的定义如下：

```
struct semid_ds
{
    struct ipc_perm sem_perm;    /* operation permission struct */
    struct sem *sem_base;        /* ptr to first semaphore in set */
    unsigned short sem_nsems;    /*numbers of semaphores in set */
    time_t sem_otime;            /* last semop time */
    time_t sem_ctime;            /* last change time */
};
```

sem 结构记录了一个信号量的信息，其定义如下：

```
struct sem
{
    unsigned short   semval;     /*semaphore value, always >= 0*/
    pid_t   sempid;              /*pid for last operarion*/
    unsigned short   semncent;   /*numbers of processes awaiting semval > curval*/
    unsigned short   semzcnt;    /*numbers of processes awaiting semval = 0*/
};
```

下面详细介绍与信号量操作有关的函数调用。

10.6.2　信号量集的相关操作

对于 System V 信号量，主要有以下 3 个 API：semget()、semop()及 semctl()。下面分别对它们进行介绍。

1. 创建或打开信号量集

使用函数 semget 可以创建或者获得一个信号量集 ID，函数原型如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int semget (key_t key, int nsems, int flag);
```

返回：若成功则返回信号量集 ID，否则返回-1。

此函数可以用于创建一个新的信号量集，或打开一个已存在的信号量集。其中，参数 key 表示所要创建或打开的信号量集对应的键值。参数 nsems 表示创建的信号量集中包含的信号量的个数，此参数只在创建一个新的信号量集时有效。参数 flag 表示调用函数的操作类型，也可用于设置信号量集的访问权限，两者通过逻辑或表示。调用函数 semget 的作用由参数 key 和 flag 决定，相应的约定与 shmget 函数类似，读者可参考 10.5.2 小节，这里不再赘述。

另外，当 semget 成功创建一个新的信号量集时，它相应的 semid_ds 结构被初始化。ipc_perm 结构中的成员被设置为相应的值，sem_nsems 设置为函数参数 nsems 的值，sem_otime 被设置为 0，sem_ctime 设置为系统当前时间。

2. 对信号量集的操作

函数 semop 用以操作一个信号量集，函数原型如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int semop (int semid, struct sembuf semoparray[ ], size_t nops);
```

返回：若成功则返回 0，若出错则返回-1。

函数中参数 semid 是一个通过 semget 函数返回的一个信号量集标识符，参数 nops 标明了参数 semoparray 所指向数组中的元素个数。参数 semoparray 是一个 struct sembuf 结构类型的数组，其中每个元素表示一个操作，由于此函数是一个原子操作，一旦执行就将执行数组中所有的操作。

结构 sembuf 用来说明所要执行的操作，其定义如下：

```
struct sembuf
{
    unsigned short    sem_num;
    short    sem_op;
    short    sem_flg;
}
```

在 sembuf 结构中，sem_num 是相对应的信号量集中的某一个资源(即指定将要进行操作的信号量)，所以其值是一个从 0 到相应的信号量集的资源总数(ipc_perm.sem_nsems)之间的整数。sem_op 指明所要执行的操作，sem_flg 说明函数 semop 的行为。sem_op 的值是一个整数，它的取值及所对应的操作说明如下：

- sem_op>0：表示进程对资源使用完毕，释放相应的资源数，并将 sem_op 的值加到信号量的值上。
- sem_op=0：进程阻塞直到信号量的相应值为 0，当信号量已经为 0，函数立即返回。如果信号量的值不为 0，则依据 sem_flg 的 IPC_NOWAIT 位决定函数动作。sem_flg 指定 IPC_NOWAIT，则 semop 函数出错返回 EAGAIN。sem_flg 没有指定 IPC_NOWAIT，则将该信号量的 semncnt 值加 1，然后进程挂起直到下述情况发生。信号量值为 0，将信号量的 semncnt 的值减 1，函数 semop 成功返回；此信号量被删除(只有超级用户或创建用户进程拥有此权限)，函数 smeop 出错返回 EIDRM；进程捕捉到信号，并从信号处理函数返回，在此情况将此信号量的 semncnt 值减 1，函数 semop 出错返回 EINTR。
- sem_op<0：请求 sem_op 的绝对值的资源。如果相应的资源数可以满足请求，则将该信号量的值减去 sem_op 的绝对值，函数成功返回。当相应的资源数不能满足请求时，这个操作与 sem_flg 有关。sem_flg 指定 IPC_NOWAIT，则 semop 函数出错返回 EAGAIN。sem_flg 没有指定 IPC_NOWAIT，则将该信号量的 semncnt 值加 1，然后进程挂起直到下述情况发生：当相应的资源数可以满足请求，该信号的值减去 sem_op 的绝对值。成功返回；此信号量被删除(只有超级用户或创建用户进程拥有此权限)，函数 smeop 出错返回 EIDRM；进程捕捉到信号，并从信号处理函数返回，在此情况将此信号量的 semncnt 值减 1，函数 semop 出错返回 EINTR。

3. 信号量集的控制

和共享内存的控制一样，信号量集也有自己的专属控制函数 semctl，函数原型如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int semctl (int semid, int semnum, int cmd, union semun arg);
```

返回：函数成功返回值大于等于 0(当 semctl 的操作为 GET 操作时返回相应的值，其余返回 0)，失败返回-1，并设置错误变量 errno。

函数中参数 semid 是一个信号量集的标识符，semnum 指定 semid 的信号集中的某一个信号量，其类似于在信号量集资源数组中的下标，用来对指定资源进行操作。参数 cmd 定义函数所要进行的操作，其取值及表达的意义与参数 arg 的设置有关，如稍后的表 10.4 所示。最后一个参数 arg 是一个联合体(union)，其定义如下：

```
union semun
{
    int val;              /* for SETVAL */
    struct semid_ds *buf ;   /* for IPC_STAT and IPC_SET */
    unsigned short array;   /* for GETALL and SETALL */
};
```

说 明

在最新版的 glibc(1.2.10)上，union semun 联合体已经被注释掉，程序员需要自己去定义这个联合体，但在老版本的库里面是可以用的。

在表 10.4 中，cmd 参数指定表中十种命令中的一种，使其在 semid 指定的信号量集合上执行此命令。其中有五条命令是针对一个特定的信号量值的，它们用 semnum 指定该集合中的一个成员。semnum 值在 0 和 nsems-1 之间(包括 0 和 nsems-1)。

表 10.4　cmd 的取值及其含义

cmd 取值	含　义
GETALL	获得信号量集 semid 中信号量的个数，并将该值赋值给无符号短整数 arg.array
GETVAL	获得信号量集 semid 中 semnum 所指定信号量的值 semval
GETNCNT	获得信号量集 semid 中等待给定信号量锁的进程数目，即 semid_ds 结构中 sem.semncnt 的值
GETPID	获得信号量集 semid 中最后一个使用 semop 函数的进程 ID，即 semid_ds 结构中 sem.sempid 的值
GETZCNT	获得信号量集 semid 中等待信号量成为 0 的进程数目，即 semid_ds 结构中 sem.semzcnt 的值
IPC_RMID	删除信号量集。此操作只能由具有超级用户的进程或信号量集拥有者的进程执行，这个操作会影响到正在使用该信号量集的进程
IPC_SET	按照参数 arp.buf 指向的结构体中的值设置此信号量集的 sem_perm.uid、sem_perm.gid 及 sem_perm.mode 的值。此操作只能由具有超级用户的进程或信号量集拥有者的进程执行
IPC_STAT	获得该信号量的 semid_ds 结构，保存在 arg.buf 指向的缓冲区
SETALL	以 arg.array 的值设置信号量集 semid 中信号量的个数
SETVAL	以 arg.val 的值设置信号量集 semid 中 semnum 所指定信号量的值 semval

程序 10.11 是关于 Linux 信号量的综合示例。本实例有两个目的，一是获取各种信号量的信息；二是利用信号量实现共享资源的申请和释放。笔者在程序中给出了详细的注释，读者可参考注释仔细分析程序的运行结果(程序运行环境是 Red Hat 9.0)。代码清单如 sem_app.c 所示。

【程序 10.11】信号量的使用举例：sem_app.c。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <errno.h>
#define SEM_PATH "/unix/my_sem"
#define max_tries 3
int semid;
int main(void)
{
int flag1,flag2,key,i,init_ok,tmperrno;
struct semid_ds sem_info;
struct seminfo sem_info2;
union semun arg;                    /*定义 union semun 联合体*/
struct sembuf askfor_res, free_res;
```

```
flag1=IPC_CREAT|IPC_EXCL|00666;
/*只执行创建信号量集的操作，00666 为信号量集的访问权限*/
flag2=IPC_CREAT|00666;
/*如果 key 为某个已存在的信号量集的键值，则执行打开这个键的操作*/
key=ftok(SEM_PATH,'a');
if(key==-1)
{
    perror("ftok error");
    exit(1);
}
init_ok=0;
semid=semget(key,1,flag1);    /*创建一个新的信号量集，其中只包含一个信号量*/
if(semid<0)
{
        tmpperrno=errno;
        perror("semget");
if(tmpperrno==EEXIST)    /*信号量集已存在，产生 EEXIST 错误，则使用 flag2*/
        {
          semid=semget(key,1,flag2);
  /*flag2 只包含了 IPC_CREAT 标志，参数 nsems(这里为 1)必须与原来的信号量数目一致*/
          arg.buf=&sem_info; /*获得信号量的 semid_ds 结构，保存在 arg.buf 指向的缓冲区*/
          for(i=0; i<max_tries; i++)
          {
            if(semctl(semid, 0, IPC_STAT, arg)==-1)
/*0 指定第一个信号量(该信号量集中唯一的一个信号量)*/
            {
perror("semctl error");
              i=max_tries;
            }
            else
            {
              if(arg.buf->sem_otime!=0)
/*sem_otime：最后一次调用 semop 的时间，信号量集创建时该值为 0*/
              {
i=max_tries;
                init_ok=1;
              }
              else
                sleep(1);
            }
        }
        if(!init_ok)
     /*do some initializing, here we assume that the first process that creates the sem will*/
     /*finish initialize the sem and run semop in max_tries*1 seconds. else it will not run*/
     /*semop any more.*/
        {
          arg.val=1;
          if(semctl(semid,0,SETVAL,arg)==-1)/*指定信号量集 semid 中信号量 0 的值 arg.val*/
            perror("semctl setval error");
        }
```

```
}
        else
        {
perror("semget error, process exit");
            exit(1);
        }
}
else    /*semid>=0; do some initializing*/
{
            arg.val=1;
            if(semctl(semid,0,SETVAL,arg)==-1)
                perror("semctl setval error");
}
/*get some information about the semaphore and the limit of semaphore in RedHat 9.0*/
        arg.buf=&sem_info;
        if(semctl(semid, 0, IPC_STAT, arg)==-1)
            perror("semctl IPC STAT");
        printf("owner's uid is %d\n",arg.buf->sem_perm.uid);/*信号量集所有者的有效用户 ID*/
        printf("owner's gid is %d\n",arg.buf->sem_perm.gid);/*所有者的有效组 ID*/
        printf("creater's uid is %d\n",arg.buf->sem_perm.cuid);/*创建者的有效用户 ID*/
        printf("creater's gid is %d\n",arg.buf->sem_perm.cgid);/*创建者的有效组 ID*/
        arg.__buf=&sem_info2;
        if(semctl(semid,0,IPC_INFO,arg)==-1)
            perror("semctl IPC_INFO");
        printf("the number of entries in semaphore map is %d \n",arg.__buf->semmap);
        printf("max number of semaphore identifiers is %d \n",arg.__buf->semmni);
        printf("mas number of semaphores in system is %d \n",arg.__buf->semmns);
        printf("the number of undo structures system wide is %d \n",arg.__buf->semmnu);
        printf("max number of semaphores per semid is %d \n",arg.__buf->semmsl);
        printf("max number of ops per semop call is %d \n",arg.__buf->semopm);
        printf("max number of undo entries per process is %d \n",arg.__buf->semume);
        printf("the sizeof of struct sem_undo is %d \n",arg.__buf->semusz);
        printf("the maximum semaphore value is %d \n",arg.__buf->semvmx);

/*now ask for available resource:*/
        askfor_res.sem_num=0;
        askfor_res.sem_op=-1;
        askfor_res.sem_flg=SEM_UNDO;
        if(semop(semid,&askfor_res,1)==-1) /*ask for resource*/
            perror("semop error");
        sleep(3); /*do some handling on the sharing resource here, just sleep on it 3 seconds*/
        printf("now free the resource\n");

/*now free resource*/
        free_res.sem_num=0;
        free_res.sem_op=1;
        free_res.sem_flg=SEM_UNDO;
        if(semop(semid,&free_res,1)==-1) /*free the resource.*/
            if(errno==EIDRM)
                printf("the semaphore set was removed\n");
```

```
/*you can comment out the codes below to compile a different version:*/
        if(semctl(semid, 0, IPC_RMID)==-1)
            perror("semctl IPC_RMID");
        else
printf("remove sem ok\n");
return 0;
}
```

使用 gcc 编译 sem_app.c，并生成可执行文件 sem_app：

```
#gcc –o sem_app sem_app.c
```

运行程序，得到输出结果：

```
#./ sem_app
owner's uid is 0
owner's gid is 0
creater's uid is 0
creater's gid is 0
the number of entries in semaphore map is 32000
max number of semaphore identifiers is 128
mas number of semaphores in system is 32000
the number of undo structures system wide is 32000
max number of semaphores per semid is 250
max number of ops per semop call is 32
max number of undo entries per process is 32
the sizeof of struct sem_undo is 20
the maximum semaphore value is 32767
now free the resource
remove sem ok
```

程序 10.11 向读者很好地演示了 Linux 下信号量的工作机制。信号量与其他进程间通信方式有所不同，它主要用于进程间同步。通常所说的 System V 信号量实际上是一个信号量的集合，可用于多种共享资源的进程间同步。每个信号量都有一个值，可以用来表示当前该信号量代表的共享资源可用(available)数量，如果一个进程要申请共享资源，那么就从信号量值中减去要申请的数目，如果当前没有足够的可用资源，进程可以睡眠等待，也可以立即返回。当进程要申请多种共享资源时，Linux 可以保证操作的原子性，即要么申请到所有的共享资源，要么放弃所有资源，这样能够保证多个进程不会造成互锁。Linux 对信号量有各种各样的限制，程序 10.11 中给出了输出结果(Red Hat 9.0 环境下)。另外，如果读者想对信号量进行更深入的理解，建议阅读 sem.h 源代码，该文件不长，但给出了信号量相关的重要数据结构。

10.7 本章小结

在实际的编程过程中，尤其是在较大型的程序项目开发过程中，进程间的通信是很关键的一部分。本章首先介绍了 Linux 进程间通信的概念，然后分别介绍了 5 种 Linux 下常用的进程

间通信机制的概念、特点和使用方法，并详细讲解创建、操作、控制和删除等操作的相关函数调用。熟练掌握这些通信机制对于方便快捷地完成进程间的通信工作是十分重要的。

实战演练

1. 在 Shell 中使用“kill-l | grep SIGRTMAX”命令查看系统的信号列表中含有 SIGRTMAX 字符串的信号名称，体会管道(|)在 shell 命令中的使用方法。

2. 一般在 root 目录下存在有 anaconda-ks.cfg、Desktop、install.log、install.log.syslog 4 个文件，使用“ls -l”命令便能看到它们的详细信息。试使用管道“|”连接“ls -l”和“grep install”两个命令，表示只查看 root 目录下的文件名中含有 install 字符串的文件信息。

3. 在 Shell 中使用“ipcs”命令查看当前系统中各种进程间通信方式的状态。

4. 编写一个程序，使用 pipe 函数创建一个匿名管道，并使用 write 向管道的一端写入数据，使用 read 函数从管道的另一端读取数据。

5. 编写一个程序，使用 mkfifo 函数创建一个命名管道，命名管道的文件名由用户从键盘输入。

6. 编写一个程序，使用 msgget 函数创建一个消息队列，并返回该消息队列的描述符。

7. 编写一个程序，使用 msgsnd 函数向消息队列中发送一个字符串数据信息“Hello! I like Linux C program!”，并通过查看消息队列的属性信息检验发送是否成功。

8. 编写一个程序，使用 msgrcv 函数从消息队列中读取数据，并将读取到的数据输出在屏幕上。

9. 编写一个程序，使用 shmget 函数创建一段共享内存，并返回该共享内存的描述符。

10. 编写一个程序，使用 write 和 read 函数向共享内存中写入和读取数据，实现不同进程间的数据信息传递。

11. 编写一个程序，使用 semget 函数创建一个信号量集，并返回该信号量集的描述符。

第11章

线程控制

线程，有时被称为轻量级进程(Lightweight Process，LWP)，是程序执行流的最小单元，一个程序可以分为多个线程，熟悉使用线程的操作和线程控制的相关系统调用，会使用户在使用Linux 系统完成各种工作的时候更加有效率。



本章内容：

◎ 线程的基本概念。

◎ Linux 下线程控制的相关函数调用。

◎ 多个线程之间的通信。

11.1 线程的基本概念

一个标准的线程由线程标识符、当前指令指针(PC)、寄存器集合和堆栈等组成；其是进程中的一个实体，是被系统独立调度和分派的基本单位。线程自己不拥有系统资源，只拥有一点在运行中必不可少的资源，但它可与同属一个进程的其他线程共享进程所拥有的全部资源。

11.1.1 Linux 线程简介

一个在第 8 章中介绍的典型的 Linux 进程可以看作其只有一个控制线程，所以这个进程在同一时刻只能做一件事情；如果使用线程则可以使得这个进程在"同一时刻"能够同时完成多个任务，这种方式有如下的优点：

- 提高应用程序响应。这对图形界面的程序尤其有意义，当一个操作耗时很长时，整个系统都会等待这个操作，此时程序不会响应键盘、鼠标、菜单的操作，而使用多线程技术，将耗时长的操作(Time Consuming)置于一个新的线程，可以避免这种尴尬的情况。
- 使多处理器系统更加有效。操作系统会保证当线程数不大于处理器数目时，不同的线程运行于不同的处理器上。
- 改善程序结构。一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半独立的运行部分，这样的程序会利于理解和修改。

线程包含了表示进程内执行环境所必需的信息，这些信息包括线程标识符(线程 ID)、一组寄存器值、栈、调度优先级和策略、信号屏蔽字、errno 变量以及线程私有数据。进程的所有信息对该进程的所有线程都是共享的，包括可执行的程序文本、程序的全局内存及堆内存、栈和文件描述符。

在 Linux 系统中很多程序需要使用多个进程来完成工作，例如许多关键的服务器应用程序有一个监听进程在不停地运行，等待客户请求到来；当一个请求到达时，这个监听进程创建(fork)一个新的进程为这个请求服务，因为对请求进行服务经常包括一些 I/O 操作，其可能阻塞进程。

在一个应用程序中使用多个进程有着一些明显的缺点：

- 由于 fork 是一个花销很大的系统调用，所以创建这些进程增加了一些基本的开销。
- 由于每个进程都有它自己的地址空间，它必须使用进程间通信的手段，如消息传递或者共享内存。
- 要把这些进程分配到不同的机器或处理器上去运行，以及在进程之间传递信息、等待进程的完成、收集结果等都需要额外的开销。

在 Linux 创建新的线程的时候，其会有一个控制线程用于控制新线程的相应工作，被称为主线程或者控制线程，如图 11.1 所示。

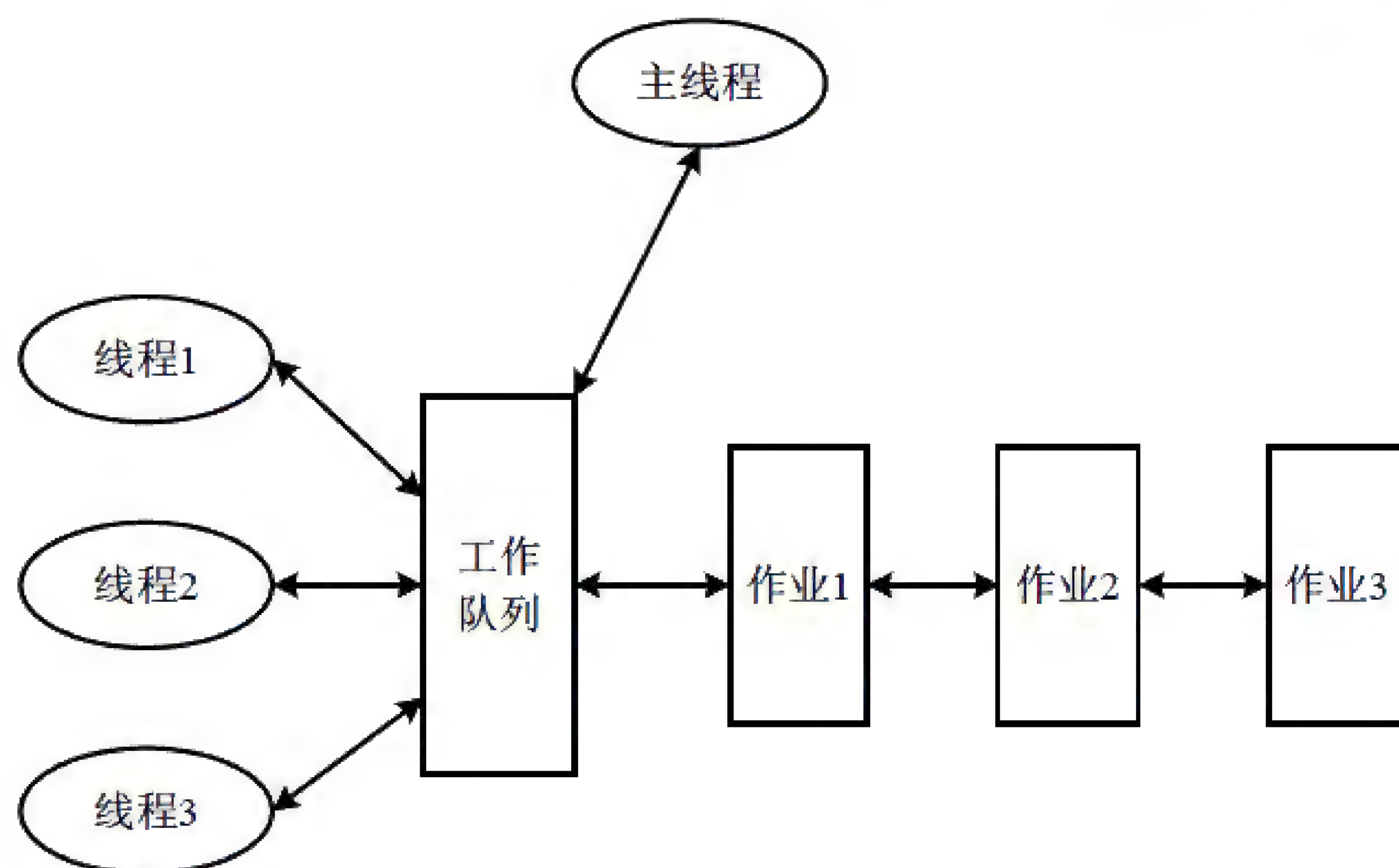


图 11.1　线程的控制

11.1.2　线程的标识符

与进程标识符类似，每一个线程都有一个在进程中唯一的线程标识符(线程 ID)，其用一个数据类型 pthread_t 来表示，该数据类型在 Linux 中其实就是一个无符号长整型数据。

Linux 提供了两个函数用于对线程标识符的操作，其标准调用格式说明如下：

```
#include <pthread.h>
pthread_t   pthread_self(void);
```

pthread_self 函数用于获得线程自身的线程标识符，其返回值是线程自身的线程标识符。

pthread_equal 函数用于比较两个线程标识符，其标准调用格式说明如下；

```
#include <pthread.h>
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

函数的两个参数分别是需要比较的两个线程的标识符，如果相等则返回一个非 0 值，否则返回 0。

11.1.3　用户态和核心态线程

用户态线程在管理上不需要内核的参与，所以通常又叫作“协作式多任务”，在进程内的这些线程统一由用户程序来切换，所以每一个线程在执行完任务后，调用任务切换功能，并向其发送信号，任务切换完成。线程对处理器资源的占用也切换到其他线程。通常，用户态线程在线程切换时要比内核线程的速度快，不过在几个比较成功的内核态线程库中，线程切换的速度也相当快。虽然用户态线程有许多灵活性和快速的特性，但是也存在一个严重的问题，即进程中的一个线程可能独占整个时间片，导致其他线程得不到处理器时间而无法运行，例如，当一个线程由于磁盘 I/O 而阻塞时，其他线程同样也不能运行。另外，用户态线程不能发挥多核心处理器机器(SMP)的性能。

内核态线程是由内核来管理的，在每一个时间片内，内核负责调度进程内的线程。由于内核参与了用户态进程的调度，所以就涉及了内核态与用户态上下文的切换。通常所说的内核态线程切换速度慢就是由于这个原因导致的。但是使用内核态线程的一个明显的好处是进程内的一个线程不会独占整个进程的处理器时间，这样，如果一个线程由于磁盘 I/O 而阻塞，其他线

程仍可以利用处理器时间运行。使用核心态线程的另外一个好处是可以充分发挥 SMP 系统的性能，而且随着系统处理器数量的增多，应用程序运行的速度明显加快。

11.1.4　线程的属性

线程的属性是通过一个名称为 pthread_attr_t 来定义的，其内部结构如下，在其中 datachstate 表示线程的拆卸状态，schedpolicy 表示线程的调度策略，schedparam 表示线程的调度参数，inheritsched 表示线程的继承性，scope 表示线程的作用域，stackaddr 表示线程堆栈的位置，stacksize 表示线程堆栈的大小。

```
typedef struct
{
    int     detachstate;
    int             schedpolicy;
    struct sched_param      schedparam;
    int             inheritsched;
    int             scope;
    size_tguardsize;
    int             stackaddr_set;
    void  *stackaddr;
    size_tstacksize;
} pthread_attr_t;
```

11.2　线程控制的相关函数

线程的控制和进程一样也是通过函数调用来实现的，本节向读者介绍线程的创建、退出和终止、阻碍和分离、取消和清理等具体操作的相关函数调用。

另外在使用 gcc 编译多线程程序时，必须与 pthread 函数库连接。在终端下编译使用下列命令：

```
gcc –lpthread
```

11.2.1　pthread_create 函数

pthread_create 函数用于创建一个新的线程，其在 Linux 函数库中的原型是：

```
#include <pthread.h>
int pthread_create (pthread_t *thread, pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);
```

返回：若成功，返回“0”；若出错则返回错误编号。

其中参数 thread 是线程的标识符，需要说明的是这个参数并不是由用户所确定的，用户只需要声明一个 pthread_t 类型的数据变量，并且将其传递给 pthread_create 函数，函数在创建新的线程的同时会将新的线程的标识符放到这个变量中。参数 attr 用于指定线程的属性，在大多数应用中可以将其设置为 NULL。参数 start_routine 用于指定开始运行的函数，新创建的线程

是从这个函数开始运行的，用户需要指定这个函数。参数 arg 则是参数 start_routine 所指定函数需要的参数，这是一个无类型指针，如果需要传递的参数不止一个，则需要将这些参数都放到一个结构中，然后将这个结构的地址传给 arg。

注 意

pthread_create 函数在调用失败之后会返回对应的错误编码，每个线程都会提供 errno 的副本。

下面是一个使用 pthread_create 函数来创建线程并且打印其线程标识符的程序。

【程序 11.1】 pthread_create 函数的使用：pthread_create.c。

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
pthread_t ntid;   //线程号
//打印标识符的函数
void   printids(const char *s)
{
            pid_t          pid;    //进程标识符
      pthread_t   tid;     //线程标识符

      pid = getpid();
      tid = pthread_self();    //分别获得进程和线程编号
      printf("%s pid %u tid %u (0x%x)\n", s, (unsigned int)pid,
         (unsigned int)tid, (unsigned int)tid);
            //打印线程和进程编号
}
//线程中开始运行的函数
void *thr_fn(void *arg)
{
      printids("new thread: ");
      return((void *)0);
}
//主函数
int    main(void)
{
      int           err;
      err = pthread_create(&ntid, NULL, thr_fn, NULL); //创建一个线程
      if (err != 0)    //如果出错则打印错误标号
            {
             printf("can't create thread: %s\n", strerror(err));
      }
            printids("main thread:");    //打印主线程号
      sleep(1);
      exit(0);
}
```

使用 gcc 编译 pthread_create.c，并且生成可执行文件 pthread_create：

```
#gcc -lpthread pthread_create.c -o pthread_create
```

运行程序，得到输出结果：

```
#./pthread_create
main thread: pid 6420 tid 3069480960 (0xb6f49000)
new thread:    pid 6420 tid 3067888752 (0xb6dc4470)
```

可以看到标识符为6420的进程创建了两个标识符分别为3069480960和3067888752的线程。

读者在阅读程序 11.2 的时候需要注意如下两点：首先这段代码需要处理主进程和新建的子进程之间的竞争，首先是主线程需要休眠，如果主线程不休眠其就可能退出，这样新线程还没运行整个进程就可能已经终止了，这种行为特征依赖于 Linux 的线程实现和调度算法；其次新线程并不是通过 thread 参数来获得相应的进程标识符，而是通过 pthread_self 函数获得，这是因为虽然新的线程会把线程标识符存放在 thread 参数中，但是由于新线程的运行时间并不确定，所以可能出现该变量还没有初始化就已经被调用的情况导致错误。

11.2.2 pthread_exit 函数

进程可以调用 exit 系列函数退出当前进程，线程也可以通过如下 3 种方式退出，在不终止整个进程的情况下停止线程的控制流。

- 线程只是从启动例程中返回，返回值是线程的退出码。
- 线程可以被同一个进程中的其他线程终止。
- 线程调用 pthread_exit 函数退出。

Linux 内核提供了 pthread_exit 函数用于主动退出线程，其在 Linux 函数库中的原型是：

```
#include <pthread.h>
void pthread_exit(void *retval);
```

pthread_exit 函数没有返回值，其参数 retval 是线程的终止状态，其与 pthread_create 函数的 start_routine 参数类似，都是由用户先指定并且传递给函数的一个参数，在 pthread_exit 函数完成之后可以调用这个参数来获得进程的退出状态。

11.2.3 pthread_ join 函数

如果当一个线程已经执行完成之后，可以被其他的线程来阻塞挂起，然后等待指定的线程调用 pthread_exit，以从启动例程中返回或者被取消，Linux 内核可以调用 pthread_join 函数来完成对线程的阻塞，其在 Linux 函数库中的原型是：

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);
```

返回：如果调用成功，函数返回 0；若出错则返回一个非 0 值。

其中参数 thread 是一个线程标识符，用于指定要等待其终止的线程；参数 retval 用于存放其他线程的返回值。

程序 11.2 演示了如何使用 phtread_join 函数来等待两个线程结束以保证完成工作之后主进程才会退出。

【程序 11.2】pthread_join 函数的使用：pthread_join.c。

```
#include <stddef.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
void print_msg(char *ptr);
//主函数
int main()
{
   pthread_t thread1, thread2;
   int i,j;
   void *retval;
   char *msg1="This is the frist thread\n";
   char *msg2="This is the second thread\n";   //存放两个字符串
   pthread_create(&thread1,NULL, (void *)(&print_msg), (void *)msg1);
   pthread_create(&thread2,NULL, (void *)(&print_msg), (void *)msg2); //创建两个线程
   pthread_join(thread1,&retval);
   pthread_join(thread2,&retval);
   return 0;
}
//打印信息函数，线程从这个函数开始运行
void   print_msg(char *ptr)
{
   int i;
   for(i=0;i<10;i++)
   printf("%s\n",ptr);   //连续输出 10 个字符串
}
```

使用 gcc 编译 pthread_join.c，并且生成可执行文件 pthread_join：

```
#gcc -lpthread pthread_join.c -o pthread_join
```

运行程序，得到输出结果：

```
#./pthread_join
This is the second thread
This is the second thread
This is the second thread
This is the second thread
This is the second thread
This is the second thread
This is the second thread
This is the second thread
This is the first thread
This is the first thread
This is the first thread
This is the first thread
This is the first thread
This is the first thread
This is the first thread
This is the first thread
This is the first thread
This is the first thread
```

11.2.4 pthread_cancel 函数

在 Linux 操作系统中，一个线程可以通过调用 pthread_cancel 函数来请求取消同一进程中的其他线程，其在 Linux 函数库中的原型是：

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

返回：如果操作成功返回 0；若失败则返回对应的错误编号。

pthread_cancel 的参数 thread 是需要取消线程的线程标识符。

11.2.5 pthread_cleanup_push 和 pthread_cleanup_pop 函数

当调用 pthread_cancel 函数取消了一个线程之后，需要调用相应的函数对进程退出之后的环境进行清理，这些函数被称为线程清理处理程序(Thread Cleanup Handler)，线程可以建立多个清理处理程序，这些函数的在 Linux 函数库中的原型是：

```
#include <pthread.h>
void pthread_cleanup_push(void (*routine)(void *),void *arg);
void pthread_cleanup_pop(int execute);
```

这两个函数都没有返回值，pthread_cleanup_push 函数将子程序 routine 连同它的参数 arg 一起压入当前线程的 cleanup 处理程序的堆栈；在当前线程调用 pthread_exit 或者是通过 pthread_cancel 终止执行时，堆栈中的处理程序将按照压栈时的相反的顺序依次调用。

而函数 pthread_cleanup_pop 从线程的 cleanup 处理程序堆栈中弹出最上面的一个处理程序并执行它。

需要注意的是，其实真正对线程执行清理工作的是在 pthread_cleanup_push 中作为参数传递进去的 routine 函数，其参数通过 arg 传递进去，其在线程执行如下动作的时候被调用：

- 调用 pthread_exit 函数的时候。
- 响应取消请求的时候。
- 用非 execute 参数调用 pthread_cleanup_pop 的时候。

如果 execute 参数被置为“0”的时候，清理函数将不会被调用，无论在哪种情况下，pthread_cleanup_pop 都将删除 pthread_cleanup_push 调用建立的清理处理程序。

程序 11.3 展示了如何使用 pthread_cleanup_push 和 pthread_cleanup_pop 函数进行线程清理。

【程序 11.3】 pthread_clean_push 和 pthread_clean_pop 函数的使用：pthread_clean.c。

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <errno.h>
//清理函数，用于输出相应的参数
void *clean(void *arg)
{
    printf("cleanup :%s   \n",(char *)arg);
    return (void *)0;
}
```

```
//线程 1 的启动函数
void *thr_fn1(void *arg)
{
    printf("thread 1 start   \n");
    //格式化清理参数
    pthread_cleanup_push( (void*)clean,"thread 1 first handler");
    pthread_cleanup_push( (void*)clean,"thread 1 second handler");
    printf("thread 1 push complete   \n");
    if(arg) //如果 arg 不为 0，返回
    {
        return((void *)1);
    }
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);//调用清理函数
    return (void *)1;
}
//线程 2 的启动函数，参考 fn1
void *thr_fn2(void *arg)
{
    printf("thread 2 start   \n");
    pthread_cleanup_push( (void*)clean,"thread 2 first handler");
    pthread_cleanup_push( (void*)clean,"thread 2 second handler");
    printf("thread 2 push complete   \n");
    if(arg)
    {
        pthread_exit((void *)2);
    }
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
    pthread_exit((void *)2);
}
//主函数
int main(void)
{
    int err;
    pthread_t tid1,tid2;   //线程标识符
    void *tret;
    //创建线程 1
    err=pthread_create(&tid1,NULL,thr_fn1,(void *)1);
    if(err!=0)   //如果创建出错
    {
        perror("create pthread 1 error\n");
        return -1;
    }
    err=pthread_create(&tid2,NULL,thr_fn2,(void *)1);

    if(err!=0)
    {
        perror("create pthread 2 error \n");
        return -1;
```

```
        }
        err=pthread_join(tid1,&tret);   //阻塞线程 1 以等待结束
        if(err!=0)
        {
            perror("join thread1 error \n");
            return -1;
        }
        printf("thread 1 exit code %d    \n",(int)tret);

        err=pthread_join(tid2,&tret);   //阻塞线程 2
        if(err!=0)
        {
            perror("join thread2 error ");
            return -1;
        }

        printf("thread 2 exit code %d    \n",(int)tret);

        return 1;
    }
```

将文件保存为 pthread_clean.c，使用 gcc 编译生成可执行文件 pthread_clean：

```
#gcc -lpthread pthread_clean.c -o pthread_clean
```

运行程序，得到输出结果：

```
#./pthread_clean
thread 2 start
thread 2 push complete
cleanup :thread 2 second handler
thread 1 start
thread 1 push complete
thread 1 exit code 1
cleanup :thread 2 first handler
thread 2 exit code 2
```

从程序的输出可以看到线程 2、线程 1 分别被取消，然后被清理。

11.2.6 pthread_detach 函数

在 Linux 中，线程一般有分离和非分离两种状态，在默认的情形下线程是非分离状态，父线程维护子线程的某些信息并等待子线程的结束，在没有显示调用 join 的情形下，子线程结束时，父线程维护的信息可能没有得到及时释放，如果父线程中大量创建非分离状态的子线程(在 LINUX 系统中使用 pthread_create 函数)，可能会出现堆栈空间不足的错误，其出错的返回值是 12。而对分离线程来说，不会有其他的线程等待它的结束，它运行结束后，线程终止，资源及时释放。

在 Linux 内核中，可以调用 pthread_deatch 函数来分离线程，其在 Linux 函数库中的原型是：

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
```

返回：如函数调用成功返回 0；若失败则返回对应的错误编号。

pthread_detach 函数的其参数是需要分离的线程标识符。

程序 11.4 展示了 pthread_detach 函数的使用方法，应用代码在主函数使用 for 循环创建了 20 个线程，然后使用 pthread_detach 函数将创建出来的线程分离；这些线程都使用相同的线程入口函数 threaddeal，该函数的用途是在屏幕上输出一个字符串显示这是当前的线程编号，该编号由 pthread_create 函数的 arg 参数传递给线程入口函数。

【程序 11.4】pthread_detach 函数的使用：pthread_detach.c。

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
//线程处理函数
void *threaddeal(void *arg)
{
  int i = *(int *)(arg);
  printf("这是第%d 个线程\n",i);
}
//主程序
int main(void)
{
    //线程 id
     pthread_t threadid;
     int j;
     //创建大量线程
     int count = 20;   //多次循环
     for(j=0 ; j < count ; j++)
     {
       //线程参数
       int * p = &(j);
       //创建线程
       int ret= pthread_create(&threadid, NULL, threaddeal, (void*)p);
       if(ret)//创建失败
       {
         printf("创建线程失败:%d\n",ret);
       }
       else//创建成功
       {
         //分离线程回收线程的 stack 占用的内存
         pthread_detach(threadid);
        }
     }
     return 0;
}
```

将文件保存为 pthread_detach.c，使用 gcc 编译生成可执行文件 pthread_detach：

```
#gcc -lpthread pthread_detach.c -o pthread_detach
```

运行程序，得到输出结果：

```
#./pthread_detach
这是第 1 个线程
这是第 2 个线程
这是第 3 个线程
这是第 6 个线程
这是第 7 个线程
这是第 6 个线程
这是第 10 个线程
这是第 12 个线程
这是第 11 个线程
这是第 4 个线程
这是第 16 个线程
这是第 16 个线程
这是第 19 个线程
这是第 16 个线程
这是第 16 个线程
这是第 16 个线程
```

从输出结果看到由于这些线程彼此之间并不同步，所以其输出序号并不是顺序排列的。

11.2.7 线程和进程操作函数对比

线程操作和进程操作有类似的地方，也有不同的地方，可以将线程和进程操作函数的对比总结如表 11.1 所示。

表 11.1 线程和进程操作函数比较

进 程 函 数	线 程 函 数	说 明
fork	pthread_create	创建一个线程或者进程
exit	pthread_exit	退出线程或者进程
waitpid	pthread_join	处理进程或者线程退出之后的状态
atexit	pthread_cleanup_push	退出控制流所调用的函数
getpid	pthread_self	获得标识符
abort	pthread_cancel	控制线程或者进程退出

11.3 线程之间的通信和同步

在第 9 章和第 10 章中向读者介绍了 Linux 的线程中通信和同步方法，而程序 11.4 也展示

了由于多个线程之间不同步导致的混乱输出结果，本节将介绍在多个线程之间的同步方法，包括互斥锁和条件变量两种。

11.3.1 互斥锁

互斥锁是一个简单的锁定命令，它可以用来锁定对共享资源的访问。对于线程来说，由于整个地址空间都是共享的资源，所以线程的任何资源都是共享的资源；互斥锁具有以下 3 个主要特点：

- 原子性：把一个互斥锁锁定为一个原子操作，这意味着操作系统(或 pthread 函数库)保证了如果一个线程锁定了一个互斥锁，没有其他线程在同一时间可以成功锁定这个互斥锁。
- 唯一性：如果一个线程锁定一个互斥锁，在它解除锁定之前，没有其他线程可以锁定这个互斥量。
- 非繁忙等待：如果一个线程已经锁定了一个互斥锁，第二个线程又试图去锁定这个互斥锁，则第二个线程将被挂起(不占用任何处理器资源)，直到第一个线程解除对这个互斥锁的锁定为止，第二个线程则被唤醒并继续执行，同时锁定这个互斥锁。

Linux 提供了一系列函数来实现互斥锁对应的操作。

1. pthread_mutex_init 函数

pthread_mutex_init 函数用来初始化一个由参数 mutex 指向的互斥锁，这个互斥锁的属性由参数 attr 指定，或者通过指定 attr 为 NULL 而使用默认的属性，其在 Linux 函数库中的原型是：

```
#include <pthread.h>
pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t recmutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
pthread_mutex_t errchkmutex = PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutex_attr *attr);
```

返回：如果执行成功，则返回 0；若失败则返回对应的错误编号。

其中 pthread_mutex_t fastmutex、pthread_mutex_t recmutex 和 pthread_mutex_t errchkmutex 这 3 个常量是常用的处理互斥锁的常量。pthread_mutex_init 函数执行成功后会把新创建的互斥锁的 ID 值放到参数 mutex 中，不会出现有多个线程同时初始化同一个互斥锁的情形，一个互斥锁在使用期间一定不会被重新初始化。

2. pthread_mutex_destroy 函数

pthread_mutex_destroy 函数用于解除由参数 mutex 指向的互斥锁的任何状态，其在 Linux 函数库中的原型是：

```
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

返回：如果执行成功返回 0；若失败则返回对应的错误编号。

需要注意的是当使用 pthread_mutex_destroy 函数解除了互斥锁状态之后，储存互斥锁的内存并不会被释放。

3. pthread_mutex_lock 函数

pthread_mutex_lock 函数可以用于锁定由参数 mutex 指向的互斥锁，其在 Linux 函数库中的原型是：

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

返回：执行成功返回 0；若失败则返回一个非 0 值。

如果函数参数中引用的 mutex 已经被锁定，那么当前调用的线程将阻塞直到互斥锁被其他线程释放(阻塞线程按照线程优先级等待)。当 pthread_mutex_lock 返回时，说明互斥锁已经被当前线程成功加锁。

4. pthread_mutex_trylock 函数

pthread_mutex_trylock 函数用于尝试给由参数 mutex 指定的互斥锁加锁，其在 Linux 函数库中的原型是：

```
#include <pthread.h>
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

返回：如果执行成功返回 0；若失败则返回一个非 0 值。

该函数是 pthread_mutex_lock 的非阻塞版本。pthread_mutex_lock 在给一个互斥锁加锁时，如果互斥锁已经被锁定，那么 pthread_mutex_lock 将一直阻塞，不会立即返回。而使用 pthread_mutex_trylock 给一个互斥锁加锁时，如果互斥锁已经被锁定，那么 pthread_mutex_trylock 调用将返回错误。否则，互斥锁将被调用者加锁。

5. pthread_mutex_unlock 函数

可以使用 pthread_mutex_unlock 函数给参数 mutex 指定的互斥锁解锁，其在 Linux 函数库中的原型是：

```
#include <pthread.h>
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

返回：如果执行成功则返回 0；若失败则返回一个非 0 值。

互斥锁必须处于加锁状态而且调用本函数的线程必须是给互斥锁加锁的同一个线程才能给互斥锁解锁。如果有其他线程在等待互斥锁，那么有核心的调度程序决定哪个线程将获得互斥锁并脱离阻塞状态。

6. 互斥锁的应用

程序 11.5 展示如何使用互斥锁来实现线程之间的同步，一个线程从共享的缓冲区中读数据，另一个线程向共享的缓冲区中写数据，使用一个互斥锁来对共享的缓冲区进行访问控制。

【程序 11.5】互斥锁的使用：pthread_mutex.c。

```
#include <stddef.h>
#include <stdio.h>
```

```
#include <unistd.h>
#include <pthread.h>
#include <stdlib.h>
#define FALSE 0
#define TRUE 1
void readfun();
void writefun();
char buffer[256];
int buffer_has_item=0;
int retflag=FALSE;
pthread_mutex_t mutex;
int main(void)
{
  pthread_t reader;
  pthread_mutex_init(&mutex,NULL);
  pthread_create(&reader,NULL,(void *)&readfun,NULL);
  writefun();
  exit(0);
}
void readfun(void)
{
  while(1)
  {
    if(retflag)
    {
      return;
    }
    pthread_mutex_lock(&mutex);
    if(buffer_has_item==1)
    {
       printf("%s",buffer);
       buffer_has_item=0;
    }
    pthread_mutex_unlock(&mutex);
  }
  return;
}

void writefun(void)
{
  int i=0;
  while(1)
  {
    if(i==10)
    {
      retflag=TRUE;
      return;
    }
    pthread_mutex_lock(&mutex);
    if(buffer_has_item==0)
```

```
        {
            sprintf(buffer,"This is %d\n",i++);
            buffer_has_item=1;
        }
        pthread_mutex_unlock(&mutex);
    }
    return;
}
```

将文件保存为 pthread_mutex.c，使用 gcc 编译生成可执行文件 pthread_mutex：

```
#gcc -lpthread pthread_ mutex.c -o pthread_ mutex
```

运行程序，得到输出结果：

```
#./pthread_mutex
This is 0
This is 1
This is 2
This is 3
This is 4
This is 5
This is 6
This is 7
This is 8
This is 9
```

可以看到和程序 11.4 不同，此时多个线程的输出不再是杂乱无章的而是遵循从小到大顺序的。

11.3.2 条件变量

在程序中使用互斥锁虽然可以解决一些资源竞争的问题，但是互斥锁只有两种状态，这使得它的用途非常有限。

Linux 还提供了条件变量作为线程的同步机制，其允许线程阻塞并等待另一个线程发送的信号。当收到信号时，阻塞的线程就被唤醒并试图锁定与之相关的互斥锁。

与互斥锁类似，Linux 提供了一些函数用于对条件变量进行操作。

1. pthread_cond_init 函数

pthread_cond_init 函数用于初始化由参数 cond 指定的条件变量，其在 Linux 函数库中的原型是：

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *cond, const pthread_cond_attr *attr);
```

返回：执行成功返回 0；若失败则返回一个非 0 值。

这个条件变量的属性由参数 attr 指定。如果参数 attr 为 NULL。那么就使用默认的属性设置。如果 pthread_cond_init 执行成功则会将新创建的条件变量的 ID 放在参数 cond 中，多线程不能同时初始化同一个条件变量。如果一个条件变量正在使用，它不能被重新初始化。

2. pthread_cond_destroy 函数

pthread_cond_destroy 函数用于来清除由参数 cond 指向的条件变量的任何状态，其在 Linux 函数库中的原型是：

```
#include <pthread.h>
int pthread_cond_destroy(pthread_cond_t *cond);
```

返回：执行成功返回 0；若失败则返回一个非 0 值。

需要注意的是，即使 pthread_cond_destroy 函数执行成功，其储存条件变量的内存空间也不被释放。

3. pthread_cond_wait 函数

使用 pthread_cond_wait 函数释放由参数 mutex 指向的互斥锁，并且使调用线程关于参数 cond 指向的条件变量阻塞。被阻塞的线程可以被 pthread_cond_signa1、pthread_cond_broadcast 或者由 fork 和传递信号引起的中断唤醒。其在 Linux 函数库中的原型是：

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

返回：执行成功返回 0；若失败则返回一个非 0 值。

需要注意的是即使返回错误信息，pthread_cond_wait 通常在互斥锁被调用线程加锁后才返回。函数将阻塞直到条件变量被信号唤醒。它在阻塞前自动释放互斥锁，在返回前再自动获得它。如果有多个线程关于条件变量阻塞，其退出阻塞状态的顺序将不确定。

4. pthread_cond_timewait 函数

pthread_cond_timedwait 函数和 pthread_cond_wait 函数的用法相似，它们的区别在于 pthread_cond_timedwait 在经过由参数 abstime 指定的时间时不阻塞。其在 Linux 函数库中的原型是：

```
#include <pthread.h>
int pthread_cond_timewait(pthread_cond_t *cond, pthread_mutex_t *mutex,
const struct timespec *abstime);
```

返回：执行成功返回 0；若阻塞条件变量的时间超过了由参数 abstime 所指定的时间，那么就返回 ETIMEOUT；若失败则返回一个非 0 值。

需要注意的是，即使是返回错误，pthread_cond_timedwait 也只在给互斥锁加锁后返回。pthread_cond_timedwait 函数将阻塞，直到条件变量获得信号或者经过由 abstime 指定的时间。

5. pthread_cond_signal 函数

pthread_cond_signal 函数用于将参数 cond 指向的条件变量阻塞的线程退出阻塞状态，其在 Linux 函数库中的原型是：

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
```

返回：执行成功返回 0；若失败则返回一个非 0 值。

需要注意的是，必须在同一个互斥锁的保护下使用 pthread_cond_signal，否则，条件变量可以在对关联条件变量的测试和 pthread_cond_wait 带来的阻塞之间获得信号，这将导致无限期的等待。如果没有一个线程关于条件变量阻塞，那么 pthread_cond_signal 无效。

6. pthread_cond_broadcast 函数

pthread_cond_broadcast 函数用于将所有关于由参数 cond 指向的条件变量阻塞的线程退出阻塞状态，其在 Linux 函数库中的原型是：

```
#include <pthread.h>
int pthread_cond_broadcast(pthread_cond_t *cond);
```

返回：执行成功返回 0；若失败则返回一个非 0 值。

这个函数将唤醒所有由 pthread_cond_wait 阻塞的线程。因为所有关于条件变量阻塞的线程都同时参与竞争，所以使用这个函数需要小心，另外需要注意的是如果没有阻塞的线程，该函数无效。

7. 条件变量的应用

在 Linux 中有一个经典的同时性编程问题叫“生产者-消费者”问题，该问题描述的是存在一个有限缓冲区和两个线程：生产者和消费者，前者分别不停地把产品放入缓冲区而后者从缓冲区中拿走产品；生产者在缓冲区满的时候必须等待，消费者在缓冲区空的时候也必须等待。此外因为缓冲区是临界资源，所以生产者和消费者之间必须互斥执行，它们之间的关系如图 11.2 所示，其本质即为两个线程的同步操作。

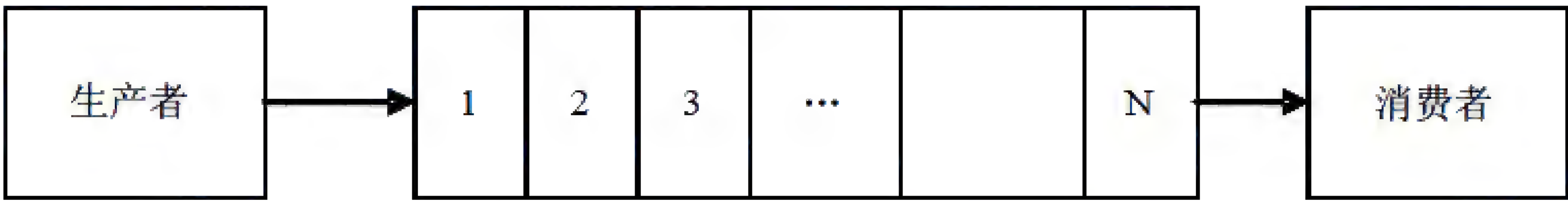


图 11.2 “生产者-消费者”问题模型

程序 11.6 展示了如何使用条件变量来同步多个线程解决“生产者-消费者”问题。

【程序 11.6】条件变量使用：pthread_cond.c。

```
#include <stdio.h>
#include <pthread.h>
#define BUFFER_SIZE 4
#define OVER (-1)
struct producers                                    //定义生产者条件变量结构
{
        int buffer[BUFFER_SIZE];                    //定义缓冲区
        pthread_mutex_t lock;                       //定义访问缓冲区的互斥锁
        int             readpos, writepos;          //读写的位置
        pthread_cond_t  notempty;                   //缓冲区有数据时的标记
        pthread_cond_t  notfull;                    //缓冲区未满的标记
};
//初始化缓冲区
```

```
void init(struct producers *b)
{
        pthread_mutex_init(&b->lock,NULL);
        pthread_cond_init(&b->notempty,NULL);
        pthread_cond_init(&b->notfull,NULL);
        b->readpos=0;
        b->writepos=0;
}
//在缓冲区中存放一个整数
void put(struct producers *b, int data)
{
        pthread_mutex_lock(&b->lock);
        //当缓冲区为满时等待
        while((b->writepos+1)%BUFFER_SIZE==b->readpos)
        {
          pthread_cond_wait(&b->notfull,&b->lock);
          //在返回之前，pthread_cond_wait 需要参数 b->lock
        }
        //向缓冲区中写数据，并将写指针向前移动
        b->buffer[b->writepos]=data;
        b->writepos++;
        if(b->writepos>=BUFFER_SIZE)
        {
         b->writepos=0;
        }
        //发送当前缓冲区中有数据的信号
        pthread_cond_signal(&b->notempty);
        pthread_mutex_unlock(&b->lock);
}
//从缓冲区中读数据并将数据从缓冲区中移走
int get(struct producers *b)
{
        int data;
        pthread_mutex_lock(&b->lock);
        //当缓冲区中有数据时等待
        while(b->writepos==b->readpos)
        {
           pthread_cond_wait(&b->notempty,&b->lock);
        }
       //从缓冲区中读数据，并将指针前移
       data=b->buffer[b->readpos];
       b->readpos++;
       if(b->readpos>=BUFFER_SIZE)
       {
         b->readpos=0;
       }
       //发送当前缓冲区未满的信号
       pthread_cond_signal(&b->notfull);
       pthread_mutex_unlock(&b->lock);
       return data;
```

```
}

struct producers    buffer;

void *producer(void *data)
{
      int n;
      for(n=0;n<10;n++)
      {
         printf("Producer: %d-->\n",n);
         put(&buffer,n);
      }
      put(&buffer,OVER);
      return NULL;
}

void *consumer(void *data)
{
      int d;
      while(1)
      {
         d=get(&buffer);
         if(d==OVER)
         {
           break;
         }
         printf("Consumer: --> %d\n",d);
      }
      return NULL;
}
//主程序
int main(void)
{
      pthread_t tha,thb;
      void *retval;
      init(&buffer);
      pthread_create(&tha,NULL,producer,0);
      pthread_create(&thb,NULL,consumer,0);
      pthread_join(tha,&retval);
      pthread_join(thb,&retval);
      return 0;
}
```

将文件保存为 pthread_cond.c，使用 gcc 编译生成可执行文件 pthread_cond：

```
#gcc -lpthread pthread_ cond.c -o pthread_ cond
```

运行程序，得到输出结果：

```
#./pthread_cond
Producer: 0-->
```

```
Producer: 1-->
Producer: 2-->
Producer: 3-->
Producer: 4-->
Consumer: --> 0
Consumer: --> 1
Consumer: --> 2
Consumer: --> 3
Consumer: --> 4
Producer: 5-->
Producer: 6-->
Producer: 7-->
Consumer: --> 5
Consumer: --> 6
Consumer: --> 7
Producer: 8-->
Producer: 9-->
Consumer: --> 8
Consumer: --> 9
```

从总输出可以看到生产者和消费者分别一次将生成的资源放入缓冲区和取出。

11.4 本章小结

本章主要讲述了 Linux 线程的概念，以及 Linux 下线程控制的相关函数调用，最后还讲到了多个线程间的同步。一个大型的程序或者进程在运行时常常会使用多个进程，尤其在目前多核处理器已经成为主流的时候，线程和多线程的使用可以大大提高程序的效率。所以了解和掌握 Linux 下线程控制的相关系统调用，以及多个线程之间相互协调运行的方法，对于程序员进行 Linux 下的程序开发是有很大益处的。

实战演练

1. 编写一个程序，使用 pthread_self 函数，获取线程自己的标识符。
2. 编写一个程序，使用 pthread_create 函数创建一个线程，在该线程中打印输出一个字符串。
3. 编写一个程序，使用 pthread_create 函数创建一个线程，然后使用 pthread_exit 函数退出。
4. 编写一个程序，使用 pthread_create 函数循环创建 5 个线程，然后每次在创建线程时候将当前循环计数器的值通过 pthread_create 函数的 arg 参数传递给新线程，在线程中打印输出该计数器的值。
5. 便携一个程序，创建 0～4 共 5 个线程，然后每个线程输出一个 hello。
6. 使用互斥锁来完成对一个公用变量的操作并且输出当前该公共变量的值。

第12章

网络编程

在第 10 章中，向读者介绍了 Linux 下常用的进程间通信方式，如管道、FIFO、消息队列、信号量和共享内存等，它们的应用局限在单一计算机内的进程间通信；而基于套接口的方式不仅可以实现单机内的进程间通信，还可以实现不同计算机进程之间的通信。本章将向读者详细阐述 Linux 网络编程的基本原理和思路，以及网络编程中常用的客户机——服务器(C/S)模式，并深入地介绍套接口(TCP 套接口、UDP 套接口和原始套接口)编程的各个环节，还给出大量的实例以帮助读者理解。



本章内容：

◎ 网络编程的基础知识。
◎ 套接口编程基础。
◎ TCP 套接口编程。
◎ UDP 套接口编程。
◎ 原始套接口编程。

12.1 网络编程的基础知识

在正式进入 Linux 网络编程的学习之前，有必要先来介绍网络编程相关的基础知识。本节内容是后面讲解 Linux 网络编程的基础，请读者务必掌握。

12.1.1 计算机网络体系结构

计算机网络是一个非常复杂的系统。为了能够使分布在不同地理且功能相对独立的计算机之间组成网络实现资源共享，计算机网络系统需要设计和解决许多复杂的问题，包括信号传输、差错控制、寻址、数据交换和提供用户接口等一系列问题。

计算机网络体系结构是为了简化这些问题的研究、设计与实现而抽象出来的一种结构模型。这种结构模型，一般采用层次模型。在层次模型中，往往将系统所要实现的复杂功能分化为若干个相对简单的细小功能，每一项分功能以相对独立的方式去实现。

1. OSI 七层模型

国际标准化组织(ISO)在 1978 年提出了开放系统互连参考模型(OSI：Open System Interconnection Reference Mode)，该模型是设计和描述网络通信的基本框架。生产厂商根据 OSI 模型的标准设计自己的产品。OSI 描述了网络硬件和软件如何以层的方式协同工作进行网络通信。

开放系统互连参考模型(OSI)采用分层的结构化技术，共分 7 层，从低到高为物理层、数据链路层、网络层、传输层、会话层、表示层、应用层。OSI 参考模型的每一层都定义了所实现的功能，完成某特定的通信任务，并只与相邻的上层和下层进行数据的交换，如图 12.1 所示。

应用层(Application Layer)
表示层(Presentation Layer)
会话层(Session Layer)
传输层(Transport Layer)
网络层(Network Layer)
数据链路层(Date Link Layer)
物理层(Physical Layer)

图 12.1　OSI 七层模型

2. OSI 参考模型各层的功能

OSI 参考模型的每一层都有它自己必须实现的一系列功能，以保证数据包能从源节点传输到目的节点。下面简单介绍 OSI 参考模型各层的功能。

(1) 物理层(Physical Layer)

物理层是 OSI 参考模型的最底层，也是 OSI 体系结构中最重要的、最基础的一层。物理层并不是指物理设备或物理媒体，而是有关物理设备通过物理媒体进行互连的描述和规定。物理

层协议定义了接口的机械特性、电气特性、功能特性、规程特性等 4 个基本特性。

物理层以比特流的方式传送来自数据链路层的数据，而不去理会数据的含义或格式。同样，它接收数据后直接传给数据链路层。也就是说，物理层只能看见 0 和 1，它没有一种机制用于确定自己所处理的比特流的具体意义，而只与数据通信的机械或电气特性有关。

(2) 数据链路层(Data Link Layer)

数据链路层是 OSI 模型的第 2 层，负责通过物理层从一台计算机到另一台计算机无差错地传输数据帧，允许网络层通过网络连接进行虚拟无差错地传输。

通常，数据链路层发送一个数据帧后，等待接收方的确认。接收方数据链路层检测帧传输过程中产生的任何问题。没有经过确认的帧和损坏的帧都要进行重传。

(3) 网络层(Network Layer)

网络层是 OSI 模型的第 3 层，负责信息寻址和将逻辑地址与名字转换为物理地址。

在网络层，数据传送的单位是包。网络层的任务就是要选择合适的路径和转发数据包，使发送方的数据包能够正确无误地按地址寻找到接收方的路径，并将数据包交给接收方。网络中两节点之间达到的路径可能有很多，应通过哪条路径才能将数据从源设备传送到所要通信的目的设备，在寻找最快捷、花费最低的路径时，必须考虑网络拥塞程度、服务质量、线路的花费和线路有效性等诸多因素。总的来说，网络层负责选择最佳路径。

网络层处于传输层和数据链路层之间，它负责向传输层提供服务，同时负责将网络地址翻译成对应的物理地址。网络层协议还能协调发送、传输及接收设备的能力不平衡的问题，如网络层对数据进行分段和重组，以使得数据的长度能够满足该网络下层数据链路层所支持的最大的数据帧(MTU)的长度。

另外，网络层还需要考虑采用不同的网络层协议的网络之间的相互连接问题，如 TCP/IP 使用的 IP 协议和 NOVELL 使用的 IPX 协议之间的相互连接。

(4) 传输层(Transport Layer)

传输层的功能是保证在不同子网的两台设备间数据包可靠、顺序、无错地传输。在传输层，数据传送的单位是段。传输层负责处理端对端通信，所谓端对端是指从一个终端(主机)到另一个终端(主机)，中间可以有一个或多个交换节点。

传输层向高层用户提供端到端的可靠的透明传输服务，为不同进程间的数据交换提供可靠的传送手段。在传输层一个很重要的工作是数据的分段和重组，即把一个上层数据分割成更小的逻辑片或物理片。换而言之，也就是发送方在传输层把上层交给它的较大的数据进行分段后分别交给网络层进行独立传输，从而实现在传输层的流量控制，提高网络资源的利用率。接收方将收到的分段的数据重组，还原成为原先完整的数据。

另外，传输层的另一个主要功能就是将收到的乱序数据包重新排序，并验证所有的分组是否都已被收到。

(5) 会话层(Session Layer)

会话层是利用传输层提供的端到端的服务，向表示层或会话用户提供会话服务。会话层的主要功能是在两个节点间建立、维护和释放面向用户的连接，并对会话进行管理和控制，保证会话数据可靠传送。

在会话层和传输层都提到了连接，那么会话连接和传输连接到底有什么区别呢？会话连接和传输连接之间有 3 种关系：

- 一对一关系，即一个会话连接对应一个传输连接。
- 一对多关系，一个会话连接对应多个传输连接。
- 多对一关系，多个会话连接对应一个传输关系。

会话过程中，会话层需要决定到底使用全双工通信还是半双工通信。如果采用全双工通信，则会话层在对话管理中要做的工作就很少；如果采用半双工通信，会话层则通过一个数据令牌来协调会话，保证每次只有一个用户能够传输数据。

会话层提供了同步服务，通过在数据流中定义检查点(Checkpoint)来把会话分割成明显的会话单元。当网络故障出现时，从最后一个检查点开始重传数据。

常见的会话层协议有结构化查询语言(SQL)、远程进程呼叫(RPC)、X-windows 系统、AppleTalk 会话协议、数字网络结构会话控制协议(DNA SCP)等。

(6) 表示层(Presentation Layer)

OSI 模型中，表示层以下的各层主要负责数据在网络中传输时不出错。但数据的传输没有出错，并不代表数据所表示的信息不会出错。表示层专门负责有关网络中计算机信息表示方式的问题。表示层负责在不同的数据格式之间进行转换操作，以实现不同计算机系统间的信息交换。

除了编码外，还包括数组、浮点数、记录、图像、声音等多种数据结构，表示层用抽象的方式来定义交换中使用的数据结构，并且在计算机内部表示法和网络的标准表示法之间进行转换。

表示层还负责数据的加密，以在数据的传输过程对其进行保护。数据在发送端被加密，在接收端解密。使用加密密钥来对数据进行加密和解密。

表示层还负责文件的压缩，通过算法来压缩文件的大小，降低传输费用。

(7) 应用层(Application Layer)

应用层是 OSI 参考模型中最靠近用户的一层，它直接与用户和应用程序打交道，负责对软件提供接口以使程序能使用网络。与 OSI 参考模型的其他层不同的是，它不为任何其他 OSI 层提供服务，而只是为 OSI 模型以外的应用程序提供服务，如电子表格程序和文字处理程序。包括为相互通信的应用程序或进程之间建立连接进行同步，建立关于错误纠正和控制数据完整性过程的协商等。应用层还包含大量的应用协议，如虚拟终端协议(Telnet)、简单邮件传输协议(SMTP)、简单网络管理协议(SNMP)、域名服务系统(DNS)和超文本传输协议(HTTP)等。

3. TCP/IP 参考模型

TCP/IP(Transmission Control Protocol/Internet Protocol)是由美国国防部创建的，所以有时又称 DoD(Department of Defense)模型，是发展至今最成功的通信协议，它被用于构筑目前最大的、开放的互联网络系统 Internet。TCP/IP 是一组通信协议的代名词，这组协议使任何具有网络设备的用户能访问和共享 Internet 上的信息，其中最重要的协议是传输控制协议(TCP)和网际协议(IP)。TCP 和 IP 是两个独立且紧密结合的协议，负责管理和引导数据报文在 Internet 上的传输。两者使用专门的报文头定义每个报文的内容。TCP 负责和远程主机的连接，IP 负责寻址，使报文被送到其该去的地方。

TCP/IP 也分为不同的层次开发，每一层负责不同的通信功能。但 TCP/IP 协议简化了层次结构，只有 4 层，由下而上分别为网络接口层、网络层、传输层、应用层，如图 12.2 所示。需

要指出的是，TCP/IP 是 OSI 模型之前的产物，所以两者间不存在严格的层对应关系。在 TCP/IP 模型中并不存在与 OSI 中的物理层与数据链路层相对应的部分，相反，由于 TCP/IP 的主要目标是致力于异构网络的相互连接，所以在 OSI 中的物理层与数据链路层相对应的部分没有进行任何限定。



图 12.2　TCP/IP 模型和 OSI 模型

4. TCP/IP 模型各层的功能

同 OSI 参考模型一样，TCP/IP 模型的每一层也都有它自己必须实现的一系列功能，下面简单介绍 TCP/IP 参考模型中各层的功能。

(1) 网络接口层

网络接口层是 TCP/IP 模型的最底层，负责接收从网络层交来的 IP 数据报，并将 IP 数据报通过底层物理网络发送出去，或者从底层物理网络上接收物理帧，抽出 IP 数据报，交给网络层。网络接口层是采用不同技术和网络硬件进行互连的，它包括属于操作系统的设备驱动器和计算机网络接口卡，以处理具体的硬件物理接口。

(2) 网络层

网络层负责独立地将分组从源主机送往目标主机，涉及为分组提供最佳路径的选择和交换功能，并使这一过程与它们所经过的路径和网络无关。TCP/IP 模型的网络层在功能上非常类似于 OSI 参考模型中的网络层，即检查网络拓扑结构，以决定传输报文的最佳路由。

(3) 传输层

传输层的作用是在源节点和目的节点的两个对等实体间提供可靠的端到端的数据通信。为保证数据传输的可靠性，传输层协议也提供了确认、差错控制和流量控制等机制。传输层从应用层接收数据，并且在必要的时候把它分成较小的单元，传递给网络层，并确保到达对方的各段信息正确无误。

(4) 应用层

应用层为用户提供网络应用，并为这些应用提供网络支撑服务，把用户的数据发送到低层，为应用程序提供网络接口。由于 TCP/IP 将所有与应用相关的内容都归为一层，所以在应用层要处理高层协议、数据表达和对话控制等任务。

5. TCP/IP 各层主要协议

TCP/IP 事实上是一个协议系列或协议簇，目前包含了 100 多个协议，用来将各种计算机和数据通信设备组成实际的 TCP/IP 计算机网络。它的特点是上下两头大而中间小，应用层和

网络接口层都有许多协议，而中间的 IP 层很小，上层的各种协议都会向下聚到一个 IP 协议中。这种很像沙漏计时器形状的 TCP/IP 协议簇表明：TCP/IP 可以为各式各样的应用提供服务(everything Over IP)，同时也可以连接到各式各样的网络上(IP Over everything)。正因为如此，因特网才发展到今天的这种全球规模。

(1) 网络接口层协议

TCP/IP 的网络接口层中包括各种物理网协议，例如 Ethernet、令牌环、帧中继、ISDN 和分组交换网 X.25 等。当各种物理网被用作传送 IP 数据包的通道时，就可以认为是属于这一层的内容。

(2) 网络层协议

网络层包括多个重要协议，主要协议有 4 个，即 IP 协议、ARP 协议、RARP 协议和 ICMP 协议。

- 网际协议(Internet Protocol，简称 IP)：IP 协议是其中的核心协议，IP 协议规定网际层数据分组的格式。
- 因特网控制消息协议(Internet Control Message Protocol，简称 ICMP)：提供网络控制和消息传递功能。
- 地址解释协议(Address Resolution Protocol，简称 ARP)：用来将逻辑地址解析成物理地址。
- 反向地址解释协议(Reverse Address Resolution Protocol，简称 RARP)：通过 RARP 广播，将物理地址解析成逻辑地址。

(3) 传输层协议

传输层的主要协议有 TCP 协议和 UDP 协议。稍后将对这两个协议进行较详细的介绍，因为本章的重点——基于套接字的网络编程，正是基于这两个协议的实现。

- 传输控制协议(Transport Control Protocol，简称 TCP)：TCP 协议是面向连接的协议，用三次握手和滑动窗口机制来保证传输的可靠性和进行流量控制。
- 用户数据报协议(User Datagram Protocol，简称 UDP)：VDP 协议是面向无连接的不可靠传输层协议。

(4) 应用层协议

应用层包括了众多的应用与应用支撑协议。常见的应用协议有文件传输协议 FTP、超文本传输协议 HTTP、简单邮件传输协议 SMTP、虚拟终端 TELNET，常见的应用支撑协议包括域名服务 DNS 和简单网络管理协议 SNMP 等。

说明

关于本节中使用到的计算机网络方面的专业名词及术语，鉴于篇幅的限制和本书的内容范畴，不再一一对它们进行解释和阐述，读者可参考计算机网络方面的书籍和资料。

12.1.2 传输控制协议 TCP

TCP 是 TCP/IP 体系中面向连接的运输层协议，它提供全双工和可靠交付的服务。一定要记住，TCP 和 UDP 的最大区别是 TCP 是面向连接的，而 UDP 是无连接的。

1. TCP 可靠数据传输技术

TCP 协议采用了许多机制来保证端到端节点之间的可靠数据传输，如采用序列号、确认重传、滑动窗口等。

首先，TCP 要为所发送的每一个报文段加上序列号，保证每一个报文段能被接收方接收，并只被正确地接收一次。

其次，TCP 采用具有重传功能的积极确认技术作为可靠数据流传输服务的基础。这里，“确认”是指接收端在正确收到报文段之后向发送端回送一个确认(ACK)信息。发送方将每个已发送的报文段备份在自己的发送缓冲区里，而且在收到相应的确认之前是不会丢弃所保存的报文段的。“积极”是指发送方在每一个报文段发送完毕的同时启动一个定时器，假如定时器的定时期满而关于报文段的确认信息尚未到达，则发送方认为该报文段已丢失并主动重发。为了避免由于网络延迟引起迟到的确认和重复的确认，TCP 规定在确认信息中捎带一个报文段的序号，使接收方能正确地将报文段与确认联系起来。

最后，采用可变长的滑动窗口协议进行流量控制，以防止由于发送端与接收端之间的不匹配而引起数据丢失。这里所采用的滑动窗口协议与数据链路层的滑动窗口协议在工作原理上是完全相同的，唯一的区别在于滑动窗口协议用于传输层是为了在端到端节点之间实现流量控制，而用于数据链路层是为了在相邻节点之间实现流量控制。TCP 采用可变长的滑动窗口，使得发送端与接收端可根据自己的 CPU 和数据缓存资源对数据发送和接收能力来进行动态调整，从而灵活性更强，也更合理。

此外，TCP 协议还采用了传输的流量控制与拥塞控制，以确保数据报文有序、无误地到达接收方。限于篇幅，这里不再详细讲解，读者可参考计算机网络的相关书籍。

2. TCP 连接的建立

TCP 连接包括建立连接、数据传输和拆除连接 3 个过程。TCP 通过 TCP 端口提供连接服务，最后通过连接服务来接收和发送数据。TCP 连接的申请、打开和关闭必须遵守 TCP 协议的规定。TCP 使用三次握手协议来建立连接，连接可以由任何一方发起，也可以由双方同时发起。一旦一台主机上的 TCP 软件已经主动发起连接请求，运行在另一台主机上的 TCP 软件就被动地等待握手。如图 12.3 所示给出了三次握手建立 TCP 连接的简单示意图。

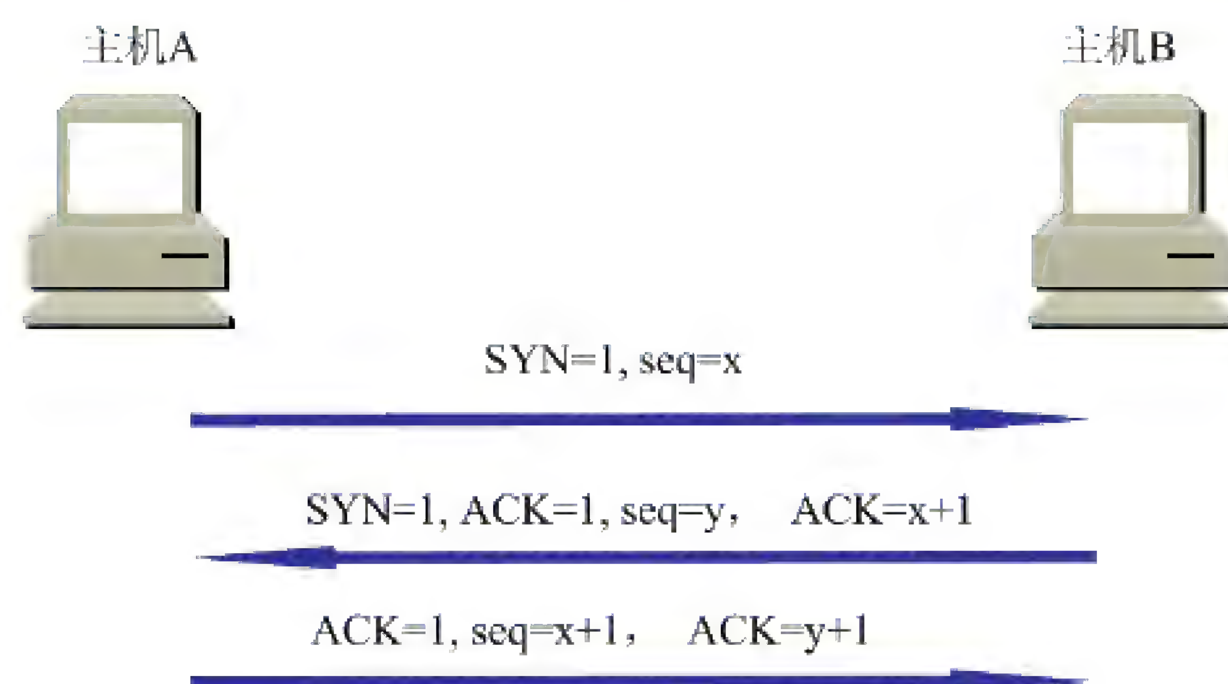


图 12.3 TCP 建立连接的三次握手

在源主机想和目的主机通信时，目的主机必须同意，否则 TCP 连接无法建立。为了确保 TCP 连接的成功建立，TCP 采用了一种称为三次握手的方式，三次握手方式使得“序号/确认

号”系统能够正常工作，从而使它们的序号达成同步。如果三次握手成功，则连接建立成功，可以开始传送数据信息。

其三次握手分别为：

(1) 源主机 A 的 TCP 向主机 B 发出连接请求报文段，其首部中的 SYN(同步)标志位应置为 1，表示想与目标主机 B 进行通信，并发送一个同步序列号 X(例：SEQ=100)进行同步，表明在后面传送数据时的第一个数据字节的序号是 X+1(即 101)。

(2) 目标主机 B 的 TCP 收到连接请求报文段后，如同意，则发回确认。在确认报中应将 ACK 位和 SYN 位置为 1。确认号应为 X+1(图 12.3 中为 101)，同时也为自己选择一个序号 Y。

(3) 源主机 A 的 TCP 收到目标主机 B 的确认后要向目标主机 B 给出确认，其 ACK 置为 1，确认号为 Y+1，而自己的序号为 X+1。TCP 的标准规定，SYN 置 1 的报文段要消耗掉一个序号。

运行客户进程的源主机 A 的 TCP 通知上层应用进程，连接已经建立。当源主机 A 向目标主机 B 发送第一个数据报文段时，其序号仍为 X+1，因为前一个确认报文段并不消耗序号。

当运行服务进程的目标主机 B 的 TCP 收到源主机 A 的确认后，也通知其上层应用进程，连接已经建立。至此建立了一个全双工的连接。

3. 关闭连接

一个 TCP 连接建立之后，即可发送数据，一旦数据发送结束，就需要关闭连接。由于 TCP 连接是一个全双工的数据通道，一个连接的关闭必须由通信双方共同完成。当通信的一方没有数据需要发送给对方时，可以使用 FIN 段向对方发送关闭连接请求。这时，它虽然不再发送数据，但并不排斥在这个连接上继续接收数据。只有当通信的对方也递交了关闭连接的请求后，这个 TCP 连接才会完全关闭。

在关闭连接时，既可以由一方发起而另一方响应，也可以双方同时发起。无论怎样，收到关闭连接请求的一方必须使用 ACK 段给予确认。实际上，TCP 连接的关闭过程也是一个三次握手的过程。

在关闭连接之前，为了确保数据正确传递完毕，仍然需要采用“三次握手”的方式来关闭连接，如图 12.4 所示。

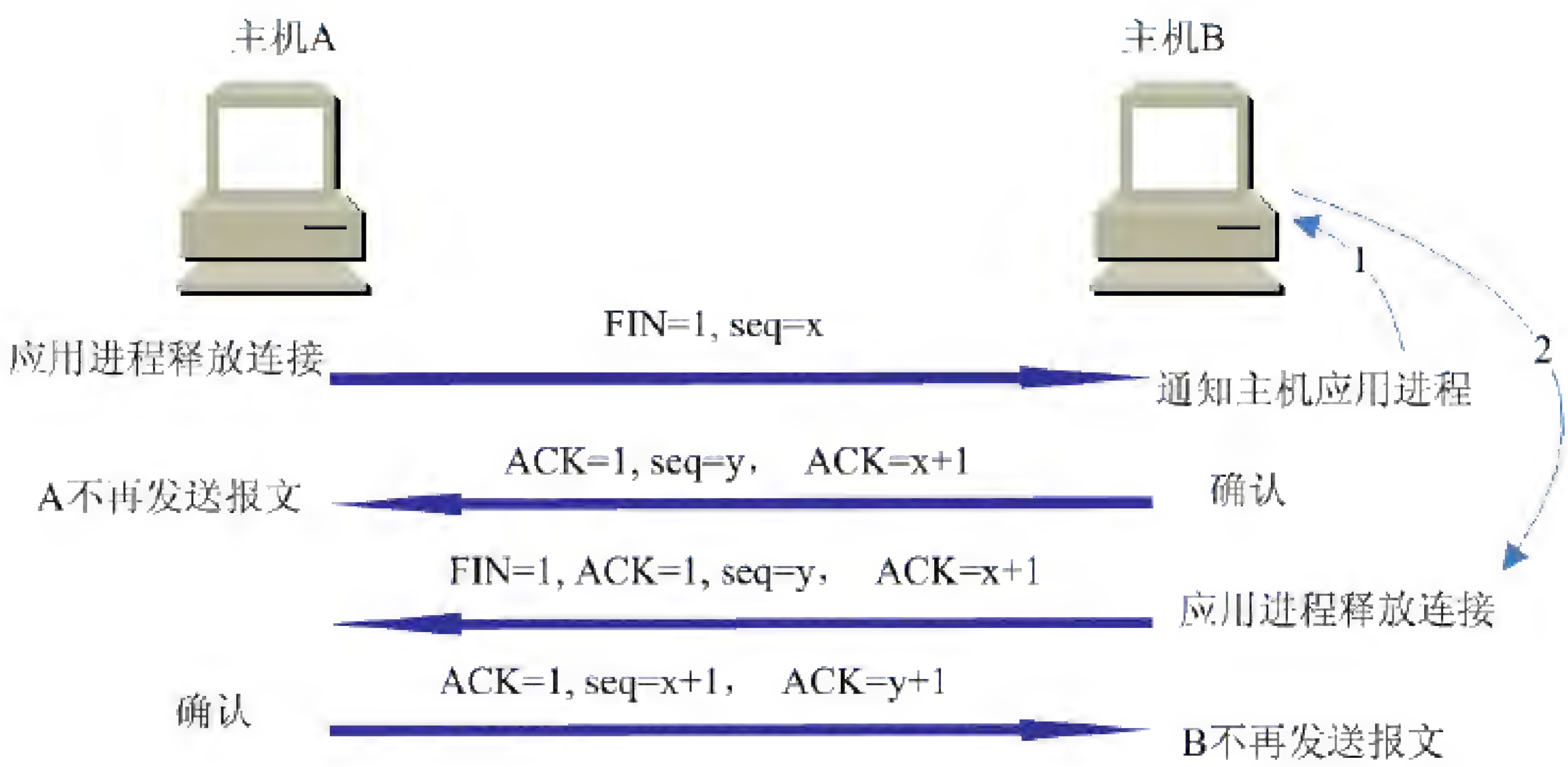


图 12.4　TCP 关闭连接的三次握手

其三次握手分别为：

(1) 源主机 A 的应用进程先向其 TCP 发出连接释放请求，并且不再发送数据。TCP 通知对方要释放从 A 到 B 这个方向的连接，将发往主机 B 的 TCP 报文段首部的终止比特 FIN 置 1，其序号 X 等于前面已传送过的数据的最后一个字节的序号加 1。

(2) 目标主机 B 的 TCP 收到释放连接通知后即发出确认，其序号为 Y，确认号为 X+1，同时通知高层应用进程，如图 12.4 中的箭头 1。这样，从 A 到 B 的连接就释放了，连接处于半关闭状态，相当于主机 A 向主机 B 说："我已经没有数据要发送了。但如果还发送数据，我仍接收。"，此后，主机 B 不再接收主机 A 发来的数据。但若主机 B 还有一些数据要发送给主机 A，则可以继续发送。主机 A 只要正确收到数据，仍应向主机 B 发送确认。

(3) 若主机 B 不再向主机 A 发送数据，其应用进程就通知 TCP 释放连接，如图 12.4 中的箭头 2。主机 B 发出的连接释放报文段必须将终止比特 FIN 和确认比特 ACK 置 1，并使其序号仍为 Y，但还必须重复上次已发送过的 ACK＝X+1。主机 A 必须对此发出确认，将 ACK 置 1，ACK＝Y+1，而自己的序号是 X+1。这样才把从 B 到 A 的反方向的连接释放掉。主机 A 的 TCP 再向其应用进程报告，整个连接已经全部释放。

12.1.3 用户数据报协议 UDP

UDP 是 TCP/IP 体系中无连接的运输层协议，与 TCP 相比，UDP 的实现要简单得多(至少在运输层可以这样说)。UDP 在传送数据之前不需要先建立连接。对方的运输层在收到 UDP 报文后，不需要给出任何确认。虽然 UDP 不提供可靠交付，但在某些情况下 UDP 是一种最有效的工作方式。

UDP 协议有如下的特点：

- UDP 传送数据前并不与对方建立连接，即 UDP 是无连接的，在传输数据前，发送方和接收方相互交换信息使双方同步。
- UDP 不对收到的数据进行排序，在 UDP 报文的首部中并没有关于数据顺序的信息(如 TCP 所采用的序号)，而且报文不一定是按顺序到达的，所以接收端无从排起。
- UDP 对接收到的数据报不发送确认信号，发送端不知道数据是否被正确接收，也不会重发数据。
- UDP 传送数据较 TCP 快速，系统开销也少。
- 由于缺乏拥塞控制(Congestion Control)，需要基于网络的机制来减小因失控和高速 UDP 流量负荷而导致的拥塞崩溃效应。换句话说，因为 UDP 发送者不能够检测拥塞，所以像使用包队列和丢弃技术的路由器这样的网络基本设备往往就成为降低 UDP 过大通信量的有效工具。

从以上特点可知，UDP 提供的是无连接的、不可靠的数据传送方式，是一种尽力而为的数据交付服务。

12.1.4 客户机/服务器模式

客户机/服务器(Client/Server)模式，即 C/S 模式，要求每个应用程序由两个部分组成：一个部分负责启动通信，另一个部分负责对它进行应答。它们通常在不同的主机上，分别称为客户

机和服务器。服务器是指那些在网络上提供服务的应用程序，客户机是指用户为了得到某种服务所需要运行的应用程序，即申请服务的程序。服务器接收网络中任何一个客户机的服务请求，完成服务后将结果返回给客户机，它们之间的关系如图 12.5 所示。

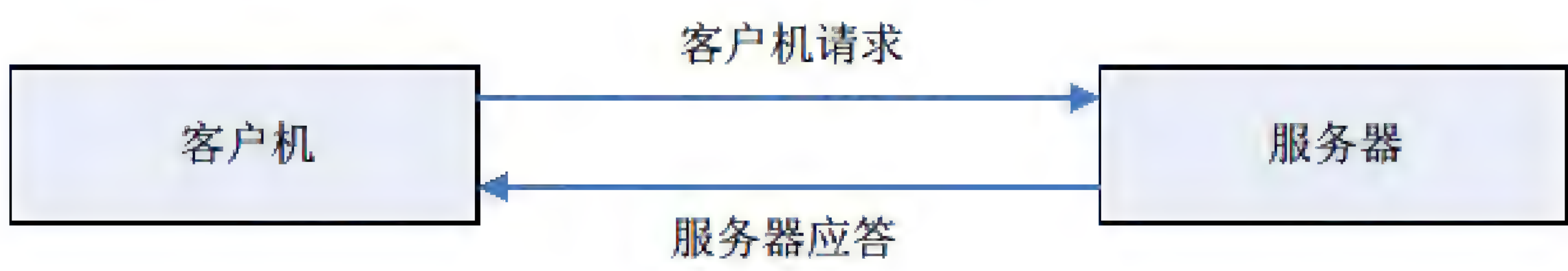


图 12.5　C/S 模式

一个服务器程序可以同时接收多个客户机的请求，当网络中的某一个客户机发送服务请求时，服务器使其在提供服务的端口排队，然后从队列中提取请求，为每个请求创建一个子进程，由子进程来处理具体的服务细节。

所以，通常情况下，服务器也包含主程序和从程序两个部分。主程序负责接收来自客户机的请求，从程序一般有几个，它们负责处理各个客户请求。

12.2 套接口编程基础

套接口是操作系统内核中的一个数据结构，它是网络中的节点进行相互通信的门户。网络编程实际上也可以称为套接口编程。在深入学习基于 TCP 和 UDP 的网络编程之前，需要先掌握好套接口的概念。本节向读者介绍套接口的概念和数据结构定义，以及 Linux 下套接口编程的一些基本函数。

12.2.1　什么是套接口

学习网络编程，无论使用哪种语言，哪种操作系统，不可避免地要遇到“socket”这个名词，它就是所说的“套接口”，或称为“套接字”。

套接口也就是网络进程的 ID。网络通信，归根到底还是进程间的通信(不同计算机上的进程间的通信)。在网络中，每一个节点(计算机或路由器)都有一个网络地址，也就是 IP 地址，两个进程通信时，首先要确定各自所在的网络节点的网络地址。但是，网络地址只能确定进程所在的计算机，而一台计算机上很可能同时运行着多个进程，所以仅凭网络地址还不能确定到底是和网络中的哪一个进程通信，因此套接口中还需要有其他的信息，也就是端口号(port)。在一台计算机中，一个端口号一次只能分配给一个进程，也就是说，在一台计算机中，端口号和进程之间是一一对应的关系。所以，使用端口号和网络地址的组合就能唯一地确定整个网络中的一个网络进程。

例如，若网络中某一计算机的 IP 为 10.92.20.160，操作系统分配给该计算机中某一应用程序进程的端口号为 1500，则图 12.6 所示就表示了此进程的套接口的组成。

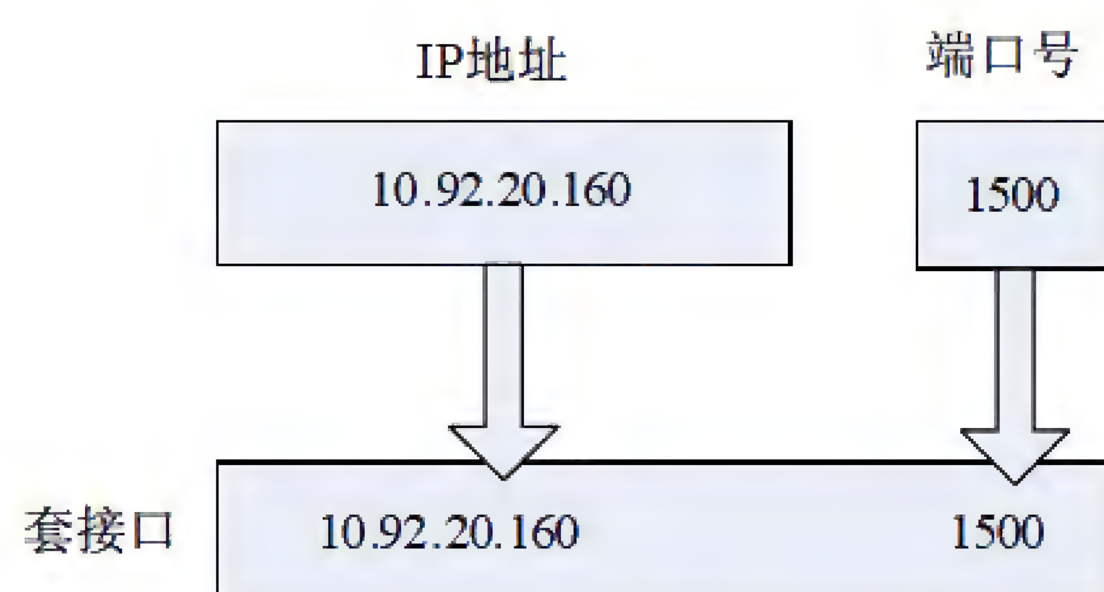


图 12.6 套接口与 IP、端口号的关系

把网络地址和端口号信息放在一个结构体中，也就是套接口地址结构。大多数的套接口函数都需要一个指向套接口地址结构的指针作为参数，并以此来传递地址信息。每个协议族都定义了它自己的套接口地址结构，套接口地址结构都以“sockaddr_”开头，并以每个协议族名中的两个字母作为结尾。

提 示

套接口在编程中，所有的函数与结构体均是小写字母，只有常量才是大写字母。

套接口有 3 种类型：流式套接口、数据报套接口和原始套接口。流式套接口也就是 TCP 套接口，用“SOCK_STREAM”表示。数据报套接口也就是 UDP 套接口(或称无连接套接口)，用“SOCK_DGRAM”表示。原始套接口用“SOCK_RAW”表示。

流式套接口定义了一种可靠的面向连接的服务，实现了无差错无重复的顺序数据传输。数据报套接口定义了一种无连接的服务，数据通过相互独立的报文进行传输，是无序的，并且不保证可靠、无差错。原始套接口允许对低层协议如 IP 或 ICMP 直接访问，主要用于新的网络协议实现的测试等。

12.2.2 端口号的概念

在网络技术中，端口(Port)大致有两种意思：一是物理意义上的端口，比如，ADSL Modem、集线器、交换机、路由器等用于连接其他网络设备的接口，如 RJ-45 端口、SC 端口等。二是逻辑意义上的端口，一般是指 TCP/IP 协议中的端口，端口号的范围从 0～65535，比如用于浏览网页服务(HTTP 协议)的 80 端口，用于 FTP 服务的 21 端口等。我们这里将要介绍的就是逻辑意义上的端口。

那么 TCP/IP 协议中的端口指的是什么呢？举个例子，如果 IP 地址唯一指定了地球上某个地理位置的一间房子，端口就是出入这间房子的门，只不过这个房子的门有 65536 个之多，端口是通过端口号来标记的，端口号是一个 16 位的整数，范围是从 0～65535。

端口号只具有本地意义，即端口号只是为了标识本地计算机上的各个进程。在因特网中不同计算机的相同端口号是没有联系的。16 bit 的端口号可允许有 64K 个端口号，这个数目对一个计算机来说是足够用的。

端口号分为两类，一类是由因特网指派名字和号码公司 ICANN 负责分配给一些常用的应用程序固定使用的“周知的端口”，其数值一般为 0～1023。表 12.1 给出了一些常见协议使用的端口号。

表 12.1　常见协议的端口号

应用程序的协议	周知的端口	应用程序的协议	周知的端口
FTP	21	TFTP	69
TELNET	23	HTTP	80
SMTP	25	SNMP	161
DNS	53	SNMP(trap)	162

“周知”就表示这些端口号是 TCP/IP 体系确定并公布的，因而是所有用户进程都知道的。当一种新的应用程序出现时，必须为它指派一个周知的端口，否则其他的应用程序进程就无法和它进行交互。

另一类则是一般端口，用来随时分配给请求通信的客户进程。

12.2.3　套接口的数据结构

下面是在网络编程中比较重要的几个数据结构，读者可以在后面介绍编程 API 部分再回过头来了解它们。

套接口的数据结构与使用它的网络有关。在 Linux 中，每一种协议都有自己的网络地址数据结构，这些结构以 sockaddr_开头，不同的是后缀表示不同的协议，如 IPv4 对应的是 sockaddr_in。这部分先介绍通用套接口的地址数据结构，因为本章注意涉及的是 IPv4 协议，然后着重介绍 sockaddr_in。

1. 通用套接口地址数据结构

由于历史的缘故，在 bind、connect 等系统调用(稍后介绍)中，特定于协议的套接口地址结构指针都要强制转换成该通用的套接口地址结构指针，以实现协议的无关性。该结构被定义在<sys/socket.h>头文件中，形式如下：

```
struct sockaddr
{
uint8_t    sa_len;
sa_family_t    sa_family;        /*address family, AF_xxx*/
char    sa_data[14];        /*14 bytes of protocol address*/
};
```

uint8_t 是 POSIX.1 要求的数据类型。在后续的介绍中，读者将会经常看到这种数据类型，它们都以“_t”结尾。为便于后续内容的介绍，将它们列于表 12.2 中。

表 12.2　POSIX.1 中的数据类型

数 据 类 型	说　　明	定义所在的头文件
int8_t	带符号的 8 位整数	<sys/types.h>
uint8_t	无符号的 8 位整数	<sys/types.h>
int16_t	带符号的 16 位整数	<sys/types.h>
uint16_t	无符号的 16 位整数	<sys/types.h>

(续表)

数 据 类 型	说　　明	定义所在的头文件
int32_t	带符号的 32 位整数	<sys/types.h>
uint32_t	无符号的 32 位整数	<sys/types.h>
sa_family_t	套接口地址结构的地址族	<sys/socket.h>
socklen_t	套接口地址结构的长度，一般为 uint32_t	<sys/socket.h>
in_port_t	TCP 或 UDP 端口号，一般为 uint16_t	<netinet/in.h>
in_addr_t	IPv4 地址，一般为 uint32_t	<netinet/in.h>

2. IPv4 套接口地址数据结构

IPv4 套接口地址数据结构以 socketaddr_in 命名，定义在<netinet/in.h>头文件中，形式如下：

```
struct socketaddr_in
{
    uint8_t   sin_len;              /*长度成员，无须设置*/
sa_family_t   sin_family;       /*套接口结构地址族，如 IPv4 为 AF_INET*/
    in_port_t   sin_port;            /* 16 位 TCP 或 UDP 端口号，网络字节顺序*/
    struct in_addr    sin_addr;
    unsigned char    sin_zero[8];    /*未用*/
};
```

结构体的成员 in_addr 也是一个结构体，定义如下所示：

```
struct in_addr
{
    in_addr_t   s_addr;      /*32 位 IPv4 地址，网络字节顺序*/
};
```

在这些结构体中，成员变量的作用与含义如下所示：

- sin_len：数据长度成员，固定长度为 16 字节，一般不要设置。
- sin_family：即为 sa_family，为调用 socket()时的 family 参数。IPv4 协议族为 AF_INET。
- sin_port：使用的端口号。
- sin_addr.s_addr：IP 地址。
- sin_zero：未使用的字段，填充为 0。

12.2.4　基本函数

本小节介绍 Linux 网络编程中常用到的基本函数，这些函数通常是为后续的套接口编程做准备工作的。读者在后面的程序实例中可以看到它们的应用。

1. 字节排序函数

数据在计算机里的存储方式可分为小端模式和大端模式两种。内存的低地址存储数据的低字节，高地址存储数据的高字节的方式称为小端模式。相反，内存的高地址存储数据的低字节，低地址存储数据高字节的方式称为大端模式。图 12.7 说明了 16 位整数(2 字节)在内存中以小端模式和大端模式的存储方式时的不同情形。

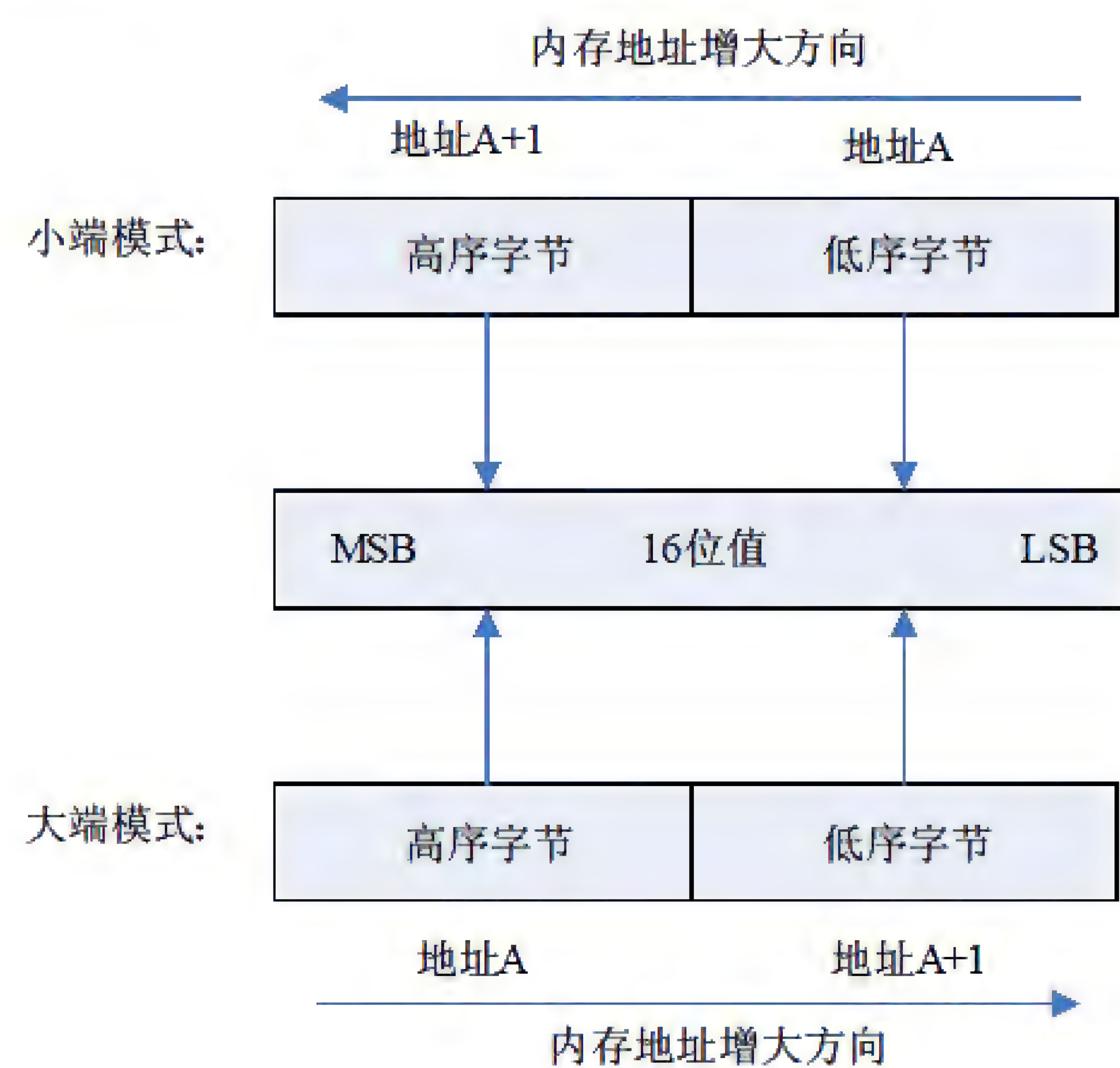


图 12.7　16 位整数在内存中的小端模式和大端模式

在网络上有着许多类型的机器，这些机器表示数据的字节顺序是不同的，比如 i386 芯片采用的是小端模式，而 alpha 芯片却相反。网际协议必须指定一个网络字节序(Network Byte Order)。

端口号和 IP 地址都是以网络字节序存储的，网络字节序使用的是大端模式。称给某个系统所采用的字节序为主机字节序，如上面所述，它可能是小端模式，也可能是大端模式。在网络协议中除了多字节数据时采用的都是网络字节序，而不是主机字节序。要把主机字节序和网络字节序相互对应起来，就要用到字节的排序函数。在 Linux 下有 4 个专门的字节排序函数：

```
#include <netinet/in.h>
uint32_t htonl (uint32_t hostlong);
uint16_t htons (uint16_t hostshort);
```

返回的是网络字节序。

```
uint32_t ntohl (uint32_t netlong);
uint16_t ntohs (uint16_t netshort);
```

返回的是主机字节序。

在这 4 个转换函数中，h 代表 host，n 代表 network，s 代表 short，l 代表 long。比如第一个函数 htonl 的意义是将本地计算机上的 long 类型数据转化为网络上的 long 类型，其他几个函数的意义就不再赘述了。

2. 字节操纵函数

在套接口编程中，经常需要一起操纵结构体中的某几个字节，这就要用到字节操纵函数了。它们的原型如下：

```
#include <strings.h>
void bzero (void *dest, size_t nbytes);
void bcopy (const void *src, void *dest, size_t nbytes);
int bcmp (const void *ptr1, const void *ptr2, size_t nbytes);
```

```
void memset (void *dest, int c, size_t len);
void memcpy (void *dest,const void *src, size_t nbytes);
int memcmp (const void *ptr1, const void *ptr2, size_t nbytes);
```

其中以 b 打头的函数由任何支持套接口函数的系统所提供，以 mem 打头的函数由 ANSI C 提供。各函数的作用和参数说明如下：

- bzero 函数：将从 dest 指定的起始地址起，长度为 nbytes(字节)的内存段设置为 0。
- bcopy 和 memcpy 函数：复制内存的数据，参数 src 指向源地址，dest 指向目的地址，nbytes 表示复制的长度。
- bcmp 和 memcmp 函数：比较内存数据的大小，参数 ptr1 和 ptr2 指向两个将要进行比较的存储区，nbytes 是以字节为单位的存储区的长度。函数的比较结果取决于第一个不相等的字节。如果 ptr1> ptr2，函数返回大于 0 的值；如果 ptr1= ptr2，函数返回 0；如果 ptr1< ptr2，函数返回小于 0 的值。
- memset 函数：用于给由 dest 指定的目标中的指定数目 len 的字节设置位值。

3. IP 地址转换函数

在 TCP/IP 网络上，我们用到的 IP 是以“.”隔开的十进制的数表示(例如 192.168.0.1)，而在套接口的数据结构中用的则是 32 位的网络字节序的二进制数值。要实现两者之间的转换，就要使用以下 3 个函数：

```
#include <arpa/inet.h>
int inet_aton (const char *straddr, struct in_addr *addrptr);
```

返回：转换成功返回 1，不成功则返回 0。

```
char *inet_ntoa (struct in_addr inaddr);
```

返回：若成功则返回指向点分十进制数串的指针，若失败则返回 NULL。

```
in_addr_t inet_addr (const char *straddr);
```

返回：若成功则返回 32 位二进制的网络字节序地址，若出错则返回 INADDR_NONE。

说 明

INADDR_NONE 是 Linux 下定义的一个常量，表示一个不存在的 IP 地址，当返回这个常数时，就说明转换出了问题。一般将这个常量定义成 255.255.255.255(它是 Internet 的有限广播地址)，用二进制表示再转换成有符号数就是-1 了。

各函数的作用和参数说明如下：

- inet_aton 函数：将点分十进制数的 IP 地址转换成为网络字节序的 32 位二进制数值。输入的点分十进制数 IP 存放在参数 straddr 中，作为返回结果的二进制数值存放在 addrptr 中。
- inet_ntoa 函数：与 inet_aton 正好相反，调用的结果作为函数的返回值返回给调用它的函数。

- inet_addr 函数：功能和 inet_aton 相同，但是结果传递的方式不同。输入的点分十进制数 IP 仍然存放在参数 straddr 中，但是结果以返回值的形式返回，函数类型为 in_addr_t，不同于 inet_aton 的整型。

4. IP 和域名的转换

在网络上标识一台计算机可以使用 IP 地址或者是域名，在网络编程中很自然的会遇到两者的转换。Linux 提供了以下两个函数实现两者的转换：

```
#include <netdb.h>
struct hostent *gethostbyname (const char *hostname);
struct hostent *gethostbyaddr (const char *addr, size_t len, int family);
```

两个函数的返回：若成功则返回一个指向 hostent 结构的指针，若失败则返回空指针 NULL，同时设置全局变量 h_errno 为相应的值。

h_errno 有如下几种可能的取值：

- HOST_NOT_FOUND：找不到主机。
- TRY_AGAIN：出错重试。
- NO_RECOVERY：不可修复性错误。
- NO_DATA：指定的名字有效，但没有记录。

这些常量都是在头文件<netdb.h>中定义的。

提 示

调用 h_strerror()函数可以得到关于 h_errno 的详细的出错信息。

其中，gethostbyname 函数实现域名或主机名到 IP 地址的转换，参数 hostname 指向存放域名或主机名的字符串。

gethostbyaddr 函数实现 IP 地址到域名或主机名的转换。参数 addr 是一个指向含有地址结构(in_addr 或 in_addr)的指针；参数 len 是此结构的大小，对于 IPv4，其值为 4，则 IPv6 的值为 16，参数 family 为协议族。

另外，结构体 struct hostent 也在<netdb.h>中定义，形式如下：

```
struct hostent
{
char *h_name;          /*主机的正式名称*/
char *h_aliases;       /*主机的别名*/
int h_addrtype;        /*主机的地址类型，IPv4 为 AF_INET*/
int h_length;          /*主机的地址长度，对于 IPv4 是 4 字节，即 32 位*/
char **h_addr_list;    /*主机的 IP 地址列表*/
};
#define h_addr h_addr_list[0]    /*主机的第一个 IP 地址*/
```

12.3 TCP 套接口编程

在前面已经向读者提到，TCP 是一个面向连接的传输层协议，在数据发送之前(即进程通

信之前)，必须先建立连接。通信完毕后，必须关闭连接。本节向读者介绍基于 TCP 协议的套接口编程。

12.3.1　TCP 套接口通信工作流程

为了实现服务器与客户机的通信，服务器和客户机都必须建立套接口。基于 TCP 传输协议的服务器与客户机间的通信工作流程可以用如图 12.8 所示的过程来描述。

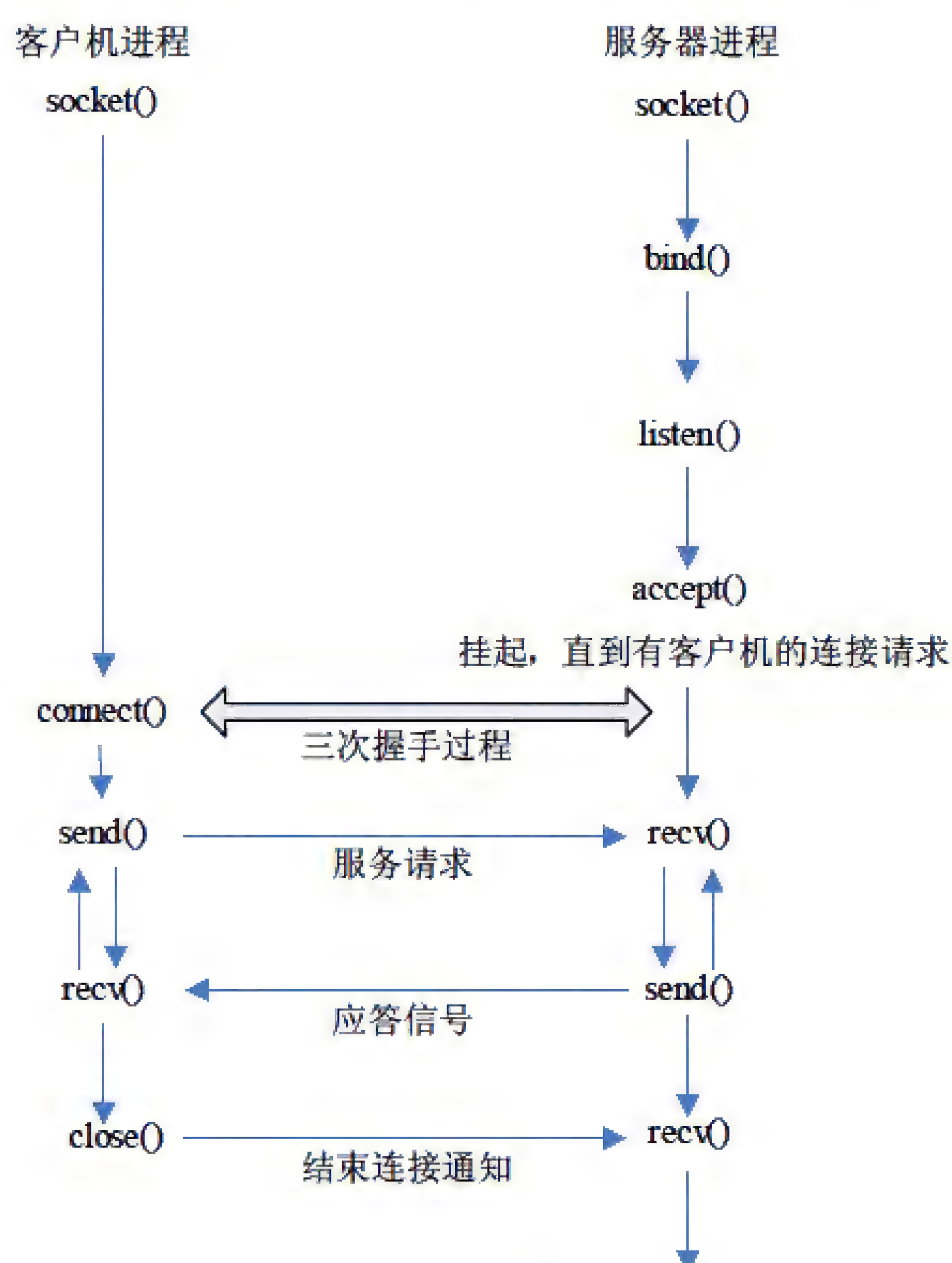


图 12.8　TCP 套接口通信工作流程

通信工作的大致流程如下：

(1) 服务器先用 socket()函数来建立一个套接口，用这个套接口完成通信的监听及数据的收发。

(2) 服务器用 bind()函数来绑定一个端口号和 IP 地址，使套接口与指定的端口号和 IP 地址相关联。

(3) 服务器调用 listen()函数，使服务器的这个端口和 IP 处于监听状态，等待网络中某一客户机的连接请求。

(4) 客户机用 socket()函数建立一个套接口，设定远程 IP 和端口。

(5) 客户机调用 connect()函数连接远程计算机指定的端口。

(6) 服务器调用 accept()函数来接受远程计算机的连接请求，建立起与客户机之间的通信连接。

(7) 建立连接以后，客户机用 write()函数(或 send()函数)向 socket 中写入数据。也可以用 read()函数(或 recv()函数)读取服务器发送来的数据。

(8) 服务器用 read()函数(或 recv()函数)读取客户机发送来的数据，也可以用 write()函数(或 send()函数)来发送数据。

(9) 完成通信以后，使用 close()函数关闭 socket 连接。

下面以这个流程为顺序，详细介绍通信过程中的各个函数调用。

1. 创建套接口

创建套接口的系统调用为 socket，它的功能是生成一个套接口描述符，函数原型如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int socket (int family, int type, int protocol);
```

返回：若成功则返回套接口描述符，若失败则返回-1。

参数 family 指明协议族，取值如 PF_UNIX(UNIX 协议族)、PF_INET(IPv4 协议)、PF_INET6(IPv6 协议)、AF_ROUTE(路由套接口)等；type 指明通信字节流类型，其取值如 SOCK_STREAM(TCP 方式)、SOCK_DGRAM(UDP 方式)、SOCK_RAW(原始套接口)、SOCK_PACKET(支持数据链路访问)等。一般来说，参数 protocol 可设置为 0，除非用在原始套接口上(原始套接口有一些特殊功能，后面还将介绍)。

说 明

socket()系统调用为套接口在 sockfs 文件系统中分配一个新的文件和 dentry 对象，并通过文件描述符把它们与调用进程联系起来。进程可以像访问一个已经打开的文件一样访问套接口在 sockfs 中的对应文件。但进程绝不能调用 open()来访问该文件(sockfs 文件系统没有可视安装点，其中的文件永远不会出现在系统目录树上)，当套接口被关闭时，内核会自动删除 sockfs 中的 inodes。

说 明

在 socket()函数的 family 参数中，IPv4 协议和 IPv6 协议分别用 PF_INET 和 PF_INET6 来表示，它们都是 sys/socket.h 头文件中定义的常量标识符。翻阅过大量书籍的读者可能会发现，在有的书籍中，使用的是 AF_INET 和 AF_INET6，甚至是两者混用。事实上，我们查看 sys/socket.h 的源代码可以知道，PF_INET 与 AF_INET，以及 PF_INET6 与 AF_INET6 是等同的。“PF”代表“Protocl Family”(协议族)，“AF”代表“Address Family”(地址族)，在不严格区分的情形下，两者是可以混用的。下文统一使用 AF_INET 或 AF_INET6。

2. 绑定端口

用 socket 函数创建一个套接口后，需要使用 bind 函数在这个套接口上绑定一个指定的端口号和 IP 地址。bind 函数的原型如下：

```
#include<sys/types.h>
#include <sys/socket.h>
int bind (int sockfd, const struct sockaddr * my_addr, socklen_t addrlen);
```

返回：若成功则返回 0，若失败则返回-1。

参数列表中，sockfd 表示已经建立的 socket 编号(描述符)，my_addr 是一个指向 sockaddr

结构体类型的指针，参数 addrlen 表示 my_addr 结构的长度，可以用 sizeof 函数来取得。

bind 函数可以把指定的 IP 与端口绑定到已经建立的 socket 上面。函数可能发生下面的错误，可以用 error 捕获发生的错误：

- EBADF：参数 sockfd 不是一个合法的 socket。
- EACCESS：权限不足。
- ENOTSOCK：参数 sockfd 是一个文件描述符，而不是 socket。

程序 12.1 是一个关于 bind 函数绑定端口的实例。程序中使用了 bind 函数在一个打开的 socket 上面绑定 IP 与端口，绑定的端口是 2345，IP 为 INADDR_ANY，表示本地计算机的默认 IP 地址。源代码如 bind.c 所示。

【程序 12.1】使用 bind 函数绑定端口：bind.c。

```
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<unistd.h>
#define PORT 2345            /*定义端口号*/

int main(void)
{
    int sockfd;                    /*定义套接口描述符*/
    struct sockaddr_in addr;        /*定义 IPv4 套接口地址数据结构 addr*/
    int addr_len = sizeof(struct sockaddr_in);
    if((sockfd = socket(AF_INET,SOCK_STREAM,0))<0)   /*建立一个 socket*/
    {
        perror("socket created error!");
        exit(1);
    }
    else            /*socket 创建成功*/
    {
        printf("socket created successfully!\n");
        printf("socket id:%d\n",sockfd);
    }
    bzero(&addr,sizeof(struct sockaddr_in));        /*清空表示地址的结构体变量*/
    addr.sin_family = AF_INET;                      /*设置 addr 的成员信息*/
    addr.sin_port = htons(PORT);
    addr.sin_addr.s_addr = htonl(INADDR_ANY); /*IP 地址设为本机 IP*/
    if(bind(sockfd, (struct sockaddr *)(&addr), sizeof(struct sockaddr))<0)
                                                /*调用 bind 函数绑定端口*/
    {
        perror("bind error!");
        exit(1);
    }
    else            /*端口绑定成功*/
    {
        printf("bind port successfully!\n");
        printf("local port:%d\n",PORT);
    }
```

```
    return 0;
}
```

使用 gcc 编译 bind.c，并生成可执行文件 bind：

```
#gcc –o bind bind.c
```

运行程序，得到输出结果：

```
#./ bind
socket created successfully!
socked id:3
bind port successfully!
local port:2345
```

3. 等待监听函数

所谓监听，指的是 socket 的端口一直处于等待的状态，监听网络中的所有客户机，耐心等待某一客户机发送请求。如果客户端有连接请求，端口就会接受这个连接。listen 函数用于实现服务器的监听等待功能，它的函数原型如下：

```
#include<sys/socket.h>
int listen (int sockfd, int backlog);
```

返回：若成功则返回 0，若失败则返回-1。

参数 sockfd 用于表示已经建立的套接口，backlog 表示能同时处理的最大连接请求数目，如果超过这个数目，客户端将会接收到 ECONNREFUSED 拒绝连接的错误。需要注意的是，listen 并未真正地接受连接，只是设置 socket 的状态为监听模式，真正接受客户端连接的是 accept 函数。通常情况下，listen 函数会在 socket，bind 函数之后调用，然后才会调用 accept 函数。

listen 函数只适用 SOCK_STREAM 或 SOCK_SEQPACKET 的 socket 类型。如果 socket 为 AF_INET，则参数 backlog 最大值可设至 128，即最多可以同时接受 128 个客户端的请求。

说明

SOCK_STREAM 是 TCP 套接口(前面已提到)，SOCK_SEQPACKET 通常用于非网络协议，例如 X.25，或是广播协议 AX.25。

listen 函数可能发生如下所示的错误，可以用 errno 来捕获发生的错误：

- EBADF：参数 sockfd 不是一个合法的 socket。
- EACCESS：权限不足。
- EOPNOTSUPP：指定的 socket 不支持 listen 模式。

4. 接受连接函数

服务器处于监听状态时，如果某时刻获得客户机的连接请求，此时并不是立即处理这个请求，而是将这个请求放在等待队列中，当系统空闲时再处理客户机的连接请求。接受连接请求的函数是 accept，函数原型如下：

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int accept (int sockfd, struct sockaddr *addr, socketlen_t *addrlen);
```

返回：若成功则返回新的套接口描述符，若失败则返回-1。

参数 sockfd 表示处于监听状态的 socket，addr 是一个 sockaddr 结构体类型的指针，系统会把远程主机的信息保存到这个指针所指向的结构体中，addrlen 表示 sockaddr 的内存长度，可以用 sizeof 函数来取得。

当 accept 函数接受一个连接时，会返回一个新的 socket 标识符，以后的数据传输与读取就是通过这个新的 socket 编号来处理，原来参数中的 socket 也可以继续使用。接受连接以后，远程主机的地址和端口信息将会保存到 addr 所指的结构体内。如果处理失败，返回值为-1。函数可能产生下面的错误，可以用 error 来捕获发生的错误：

- EBADF：参数 sockfd 不是一个合法的 socket。
- EFAULT：参数 addr 指针指向无法存取的内存空间。
- ENOTSOCK：参数 sockfd 为一文件描述符，而不是一个 socket。
- EOPNOTSUPP：指定的 socket 不是 SOCK_STREAM。
- EPERM：防火墙拒绝这个连接。
- ENOBUFS：系统的缓冲内存不足。
- ENOMEM：核心内存不足。

程序 12.2 演示了 listen 函数和 accept 函数的用法。在程序中，先建立一个 socket，然后用 bind 函数在这个 socket 上面绑定一个端口，然后使用 listen 函数使这个端口处于监听状态。当有连接请求时，accept 函数会产生一个新的 socket，然后输出提示信息。程序的代码如 lisn_acp.c 所示。

【程序 12.2】 listen 函数和 accept 函数的使用：lisn_acp.c。

```
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<unistd.h>
#define PORT 2345                          /*定义端口号*/

int main(void)
{
    int sockfd,newsockfd;                  /*定义两个套接口描述符*/
    struct sockaddr_in addr;               /*定义 IPv4 套接口地址数据结构 addr*/
    int addr_len = sizeof(struct sockaddr_in);
    if((sockfd = socket(AF_INET,SOCK_STREAM,0))<0)    /*建立一个 socket*/
    {
        perror("socket created error!");
        exit(1);
    }
    else        /*socket 创建成功*/
    {
        printf("socket created successfully!\n");
        printf("socket id:%d\n",sockfd);
    }
    bzero(&addr,sizeof(struct sockaddr_in));          /*清空表示地址的结构体变量*/
```

```
    addr.sin_family = AF_INET;                /*设置 addr 的成员信息*/
    addr.sin_port = htons(PORT);
    addr.sin_addr.s_addr = htonl(INADDR_ANY); /*IP 地址设为本机 IP*/
    if(bind(sockfd, (struct sockaddr *)(&addr), sizeof(struct sockaddr))<0)
                                              /*调用 bind 函数绑定端口*/
    {
        perror("bind error!");
        exit(1);
    }
    else          /*端口绑定成功*/
    {
         printf("bind port successfully!\n");
         printf("local port:%d\n",PORT);
    }
    if(listen(sockfd,5)<0) /*调用 listen 函数监听端口号，能同时处理的最大连接请求数为 5*/
    {
        perror("listen error!");
        exit(1);
    }
    else
    {
        printf("listening......\n");      /*监听中……*/
    }
    if((newsockfd = accept(sockfd,(struct sockaddr *)(&addr),&addr_len))<0)
/*调用 accept 接受一个连接请求*/
    {
        perror("accept error!");
    }
    else
    {
        printf("accepted a new connection.\n");     /*已接受连接请求*/
        printf("new socket id:%d\n", newsockfd);    /*新的套接口 ID*/
    }
    return 0;
}
```

使用 gcc 编译 lisn_acp.c，并生成可执行文件 lisn_acp：

```
#gcc –o lisn_acp lisn_acp.c
```

运行程序，得到输出结果：

```
#./ lisn_acp
socket created successfully!
socket id:3
bind port successfully!
local port:2345
listening......
```

从中可以看到，程序运行到这里停止了，并一直在这里等待，说明本地计算机的 2345 号端口正处于监听的状态，等待本机上的连接服务请求。此时打开浏览器，在浏览器的地址栏中

输入下列形式的地址：

```
http://192.168.1.101:2345/
```

其中，192.168.1.101 为笔者个人计算机的 IP 地址。按“Enter”键，这样浏览器会请求连接本地计算机上的 2345 号端口。

浏览器会显示无法打开这个网页(这是正常的，因为本机并没有制作 Web 主页)，但此时发现在 shell 终端中显示了如下的结果：

```
accepted a new connection.
new socket id:4
```

表明程序已经接受了这个连接，并创建了一个新的套接口(ID 为 4)，然后退出了程序。

5. 请求连接函数

所谓请求连接，是指在客户机向服务器发送信息之前，需要先发送一个连接请求，请求与服务器建立 TCP 通信连接(参考 12.1.2 小节)。connect 函数可以完成这项功能，函数原型如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int connect (int sockfd, const struct sockaddr * serv_addr, int addrlen);
```

返回：如果连接成功，返回值为 0，连接失败则返回-1。

参数 sockfd 表示已经建立的 socket，serv_addr 是一个结构体指针，指向一个 sockaddr 结构体，这个结构体存储着远程服务器的 IP 与端口号信息，addrlen 表示 sockaddr 结构体的内存长度，可以用 sizeof 函数来获取。

connect 函数会将本地的 socket 连接到 serv_addr 所指定的服务器 IP 与端口号上去。函数可能发生下面的错误，可以用 error 来捕获发生的错误：

- EBADF：参数 sockfd 不是一个合法的 socket。
- EFAULT：参数 serv_addr 指针指向了一个无法读取的内存空间。
- ENOTSOCK：参数 sockfd 是文件描述符，而不是一个正常的 socket。
- EISCONN：参数 sockfd 的 socket 已经处于连接状态。
- ECONNREFUSED：连接要求被服务器拒绝。
- ETIMEDOUT：需要的连接操作超过限定时间仍未得到响应。
- ENETUNREACH：无法传送数据包至指定的主机。
- EAFNOSUPPORT：sockaddr 结构的 sa_family 不正确。
- EALREADY：socket 不能阻断，但是以前的连接操作还未完成。

程序 12.3 是一个 connect 函数的应用实例，在程序中连接到远程服务器。在程序 12.3 中连接的远程服务器是百度网站，域名是 www.baidu.com。可以在终端中输入下面的命令来取得这个域名的 IP 地址。

```
# ping www.baidu.com
```

显示结果中的第一行如下所示：

```
PING www.a.shifen.com (119.75.213.61) 56(84) bytes of data.
```

可知百度网站的服务器 IP 地址是 119.75.213.61。网站服务的端口是 80(HTTP)。程序的代码如 remote_connect.c 所示。

【程序 12.3】使用 connect 函数连接远程服务器：remote_connect.c。

```
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/socket.h>

#define PORT 80                          /*定义一个端口号*/
#define REMOTE_IP "119.75.213.61"        /*定义一个 IP 地址*/

int main(void)
{
    int sockfd;
    struct sockaddr_in addr;        /*定义 IPv4 套接口地址数据结构 addr*/
    if( (sockfd = socket(AF_INET,SOCK_STREAM,0))<0 )   /*建立一个 socket*/
    {
        perror("socket created error!");
        exit(1);
    }
    else          /*socket 创建成功*/
    {
        printf("socket created successfully!\n");
        printf("socket id:%d\n",sockfd);
    }
    bzero(&addr,sizeof(struct sockaddr_in));       /*清空表示地址的结构体变量*/
    addr.sin_family = AF_INET;                     /*设置 addr 的成员信息*/
    addr.sin_port=htons(PORT);
    addr.sin_addr.s_addr=inet_addr(REMOTE_IP);
    if(connect(sockfd, (struct sockaddr *)(&addr), sizeof(struct sockaddr))<0)
                                      /*调用 connect 连接到远程服务器*/
    {
        perror("connect error!");
        exit(1);
    }
    else           /*连接成功，输出相关信息*/
    {
        printf("connected successfully!\n");
    }
return 0;
}
```

使用 gcc 编译 remote_connect.c，并生成可执行文件 remote_connect：

```
#gcc –o remote_connect remote_connect.c
```

运行程序，得到输出结果：

```
#./ remote_connect
socket created successfully!
socket id:3
connected successfully!
```

6. 数据发送函数

建立套接口并完成通信连接以后，可以把信息传送到远程主机上，这个过程就是信息的发送。而对于远程主机发送来的信息，本地主机需要进行接收处理。下面开始讲述这种面向连接的套接口信息发送与接收操作。

用 connect 函数连接到远程计算机以后，可以用 send 函数将信息发送到对方的计算机。当然，对方计算机也可以用 send 函数将应答信息发送给请求服务的本地主机，通信是双向的，并且通信的双方是对等的(只不过在一次通信过程中，一个为客户机，一个为服务器)。send 函数原型如下所示：

```
#include <sys/types.h>
#include <sys/socket.h>
int send (int sockfd, const void * msg, int len, unsigned int flags);
```

返回：若成功则返回已发送的字符数，若失败则返回-1。

参数 sockfd 表示已经建立的 socket，msg 是需要发送数据的指针，len 表示需要发送数据的长度，可以用 sizeof 函数来取得，参数 flags 一般设置为 0，其他可能的赋值与含义如表 12.3 所示。

表 12.3　flags 的取值及其含义

flags 取值	含　义
MSG_OOB	传送的数据以带外的(out-of-band)方式送出
MSG_DONTROUTE	取消路由表查询
MSG_WAITALL	设置数据的传送为不可阻断的传输，除非有错误或信号产生
MSG_NOSIGNAL	此传输不可被 SIGPIPE 信号中断

如果发送数据成功，函数会返回已经传送的字符个数，否则会返回-1。函数可能发生下面这些错误，可以用 errno 来捕获函数的错误：

- EBADF：参数 sockfd 不是一个正确的 socket。
- EFAULT：参数中的指针指向了不可读取的内存空间。
- ENOTSOCK：参数 sockfd 是一个文件描述符，而不是一个 socket。
- EINTR：发送进程被信号中断。
- EAGAIN：此操作会中断进程，但 socket 不允许被中断。
- ENOBUFS：系统的缓冲内存不足。
- ENOMEM：核心内存不足。
- EINVAL：传给系统调用的参数不正确。

7. 数据接收函数

函数 recv 可以接收远程主机发送来的数据，并把这些数据保存到一个数组中。函数原型如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int recv (int sockfd, void *buf, int len, unsigned int flags);
```

返回：若成功则返回接收到的字符数，若失败则返回-1。

参数 sockfd 表示已经建立的 socket，buf 是一个指针，指向一个数组，接收到的数据会保存到这个数组中，len 表示数组的长度，可以用 sizeof 函数来取得，flags 一般设置为 0，其他可能的赋值与含义如表 12.4 所示。

表 12.4　flags 的取值及其含义

flags 取值	含　义
MSG_OOB	接收以带外(out-of-band)送出的数据
MSG_PEEK	返回来的数据并不会在系统内删除，如果再调用 recv 时会返回相同的数据内容
MSG_WAITALL	强迫接收到 len 大小的数据后才能返回，除非有错误或信号产生
MSG_NOSIGNAL	此操作不能被 SIGPIPE 信号中断

recv 函数如果接收到数据，会把这些数据保存在 buf 指针指向的内存中，然后返回接收到字符的个数。如果发生错误则会返回-1。函数可能发生下面这些错误，可以用 errno 来捕获错误：

- EBADF：参数 sockfd 不是一个合法的 socket。
- EFAULT：参数中的指针指向了无法读取的内存空间。
- ENOTSOCK：参数 sockfd 是文件描述符，而不是一个 socket。
- EINTR：进程被信号中断。
- EAGAIN：此动作会阻断进程，但参数 sockfd 的 socket 不可阻断。
- ENOBUFS：系统的缓冲内存不足。
- ENOMEM：核心内存不足。
- EINVAL：参数不正确。

程序 12.4 讲解了一个 recv 函数的使用实例。在程序 12.4 中，调用 connect 连接到 GNU 的 FTP 服务器，然后用 recv 函数获得 FTP 服务器返回的信息。GNU 的 FTP 服务器域名为 ftp://ftp.gnu.org/。在终端中输入下面的命令，取得这个域名的 IP 地址。

```
ping ftp.gnu.org
```

终端中显示的第一行结果如下所示：

```
PING ftp.gnu.org (140.186.70.20) 56(84) bytes of data.
```

由此可知该 FTP 服务器的 IP 地址是 140.186.70.20。FTP 服务的端口号是 21。程序的源代

码如 remote_recv.c 所示。

【程序 12.4】使用 recv 函数获得远程服务器的返回信息：remote_recv.c。

```
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/socket.h>
#define PORT 21                          /*定义端口号*/
#define REMOTE_IP "140.186.70.20"        /*定义 IP 地址*/

int main(void)
{
    int sockfd;
    struct sockaddr_in addr;            /*定义 IPv4 套接口地址数据结构 addr*/
    char buf[256];                      /*用来接收数据的缓冲区*/
    if((sockfd=socket(AF_INET,SOCK_STREAM,0)) < 0)    /*建立一个 socket*/
    {
        perror("socket created error!");
        exit(1);
    }
    else            /*socket 创建成功*/
    {
        printf("socket created successfully!\n");
        printf("socked id: %d \n", sockfd);
    }
    bzero(&addr,sizeof(struct sockaddr_in));        /*清空表示地址的结构体变量*/
    addr.sin_family =AF_INET;                       /*设置 addr 的成员信息*/
    addr.sin_port=htons(PORT);
    addr.sin_addr.s_addr=inet_addr(REMOTE_IP);
if(connect(sockfd, (struct sockaddr *)(&addr), sizeof(struct sockaddr))<0)
                                                /*调用 connect 连接到远程服务器*/
    {
        perror("connect error!");
        exit(1);
    }
    else              /*连接成功，输出相关信息*/
    {
        printf("connected successfully!\n");
        printf("remote ip:%s\n",REMOTE_IP);
        printf("remote port:%d\n",PORT);
    }
    recv(sockfd ,buf,sizeof(buf),0);        /*接收信息*/
    printf("%s\n",buf);                     /*输出接收到的信息*/
    return 0;
}
```

使用 gcc 编译 remote_recv.c，并生成可执行文件 remote_recv：

```
#gcc –o remote_recv remote_recv.c
```

运行程序，得到输出结果：

```
#./ remote_recv
socket created successfully!
socked id: 3
connected successfully!
remote ip:140.186.70.20
remote port:21
220 GNU FTP server ready.
```

程序的运行结果表明程序已经正确连接到了 GNU 的 FTP 服务器，并且服务器返回了一段欢迎信息“GNU FTP server ready.”。

8. write 与 read 函数

在本书的第 6 章中曾向读者介绍了 write 与 read 函数，write 函数用来向文件中写入数据，read 函数用来从文件中读取数据。在网络编程中，当 socket 建立连接以后，向这个 socket 中写入数据即表示向远程主机传送数据，从 socket 中读取数据则相当于接受远程主机传送过来的数据。所以，write 与 read 函数也可以用于套接口的编程中，它们的作用分别与 send 和 recv 函数类似。这里再次给出它们的原型：

```
#include <unistd.h>
ssize_t write (int fd, const void *buf, size_t count);
ssize_t read (int fd, void *buf, size_t count);
```

返回：若成功则返回已写入或已读取的字节数，若出错则返回-1。

在参数列表中，fd 表示已经建立的 socket，buf 是指向一段内存的指针，count 表示 buf 指向内存的长度。read 函数读取字节时，会把读取的内容保存到 buf 指向的内存中，然后返回读取到字节的个数。使用 write 函数传输数据时，会把 buf 指针指向的内存中的数据发送到 socket 连接的远程主机，然后返回实际发送字节的个数。

程序 12.5 是一个关于利用 read 函数读取数据的实例。在程序中，监听一个端口，如果有客户端请求连接这个端口，则服务器接受这个连接，然后调用 read 函数读取远程主机发送来的数据，最后输出这些数据。程序的代码见 remote_read.c。

【程序 12.5】使用 read 函数读取远程服务器的返回信息：remote_read.c。

```
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<unistd.h>
#define PORT 5566                    /*定义端口号*/

int main(void)
{
    int sockfd,newsockfd;            /*定义相关的变量*/
```

```
    struct sockaddr_in addr;          /*定义 IPv4 套接口地址数据结构 addr*/
    int addr_len = sizeof(struct sockaddr_in);
    char msgbuf[256];                 /*发送和接收数据缓冲区*/
    if ((sockfd = socket(AF_INET,SOCK_STREAM,0))<0)   /*建立一个 socket*/
    {
        perror("socket created error! ");
        exit(1);
    }
    else        /*socket 创建成功*/
    {
        printf("socket created successfully!\n");
        printf("socked id: %d \n",sockfd);
    }
    bzero(&addr,sizeof(struct sockaddr_in));   /*清空表示地址的结构体变量*/
    addr.sin_family =AF_INET;                  /*设置 addr 的成员信息*/
    addr.sin_port = htons(PORT);
    addr.sin_addr.s_addr = htonl(INADDR_ANY);   /*IP 地址设为本机 IP*/
    if(bind(sockfd, (struct sockaddr *)(&addr), sizeof(struct sockaddr))<0)
                                               /*调用 bind 函数绑定端口*/
    {
        perror("bind error!");
        exit(1);
    }
    else        /*端口绑定成功*/
    {
        printf("bind port successfully!\n");
        printf("local port:%d\n",PORT);
    }
    if(listen(sockfd,5)<0) /*调用 listen 函数监听端口号，能同时处理的最大连接请求数为 5*/
    {
        perror("listen error!");
        exit(1);
    }
    else
    {
        printf("listening......\n");      /*端口监听中……*/
    }
    if((newsockfd = accept(sockfd,(struct sockaddr *)(&addr),&addr_len))<0)
/*调用 accept 接受一个连接请求*/
    {
        perror("accept error!");
    }
    else                          /*连接成功，输出结果*/
    {
        printf("connect from %s\n",inet_ntoa(addr.sin_addr));
        if(read(newsockfd,msgbuf,sizeof(msgbuf))<=0)     /*调用 read 接收信息*/
        {
            perror("accept error!");
        }
        else
```

```
            {
                printf("message:\n%s \n",msgbuf);       /*输出接收到的信息*/
            }
        }
    return 0;
}
```

使用 gcc 编译 remote_read.c，并生成可执行文件 remote_read：

```
#gcc –o remote_read remote_read.c
```

运行程序，得到输出结果：

```
#./remote_read
socket created successfully!
socked id: 3
bind port successfully!
local port:5566
listening......
```

与我们在程序 12.2 中看到的一样，程序运行到这里停止运行了，并一直在这里等待，说明本地计算机的 5566 号端口正处于监听的状态，等待本机上的连接服务请求。此时打开浏览器，在浏览器的地址栏中输入下列形式的地址：

```
http://192.168.1.101:5566/
```

同样，192.168.1.101 为笔者个人计算机的 IP 地址。按“Enter”键，这样浏览器会请求连接本地计算机上的 5566 号端口。

浏览器会显示无法打开这个网页(这是正常的，因为本机并没有制作 Web 主页)，但此时发现在 shell 终端中显示了如下的结果(这些代码是浏览器向本机的 5566 号端口请求打开网页的数据报)：

```
connect from 192.168.1.88
message:
GET / HTTP/1.1
Host: 192.168.1.88:5566
User-Agent: Mozilla/5.0 (X11; U; Linux i686; zh-CN; rv:1.2.1) Gecko/20030225
Accept:text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,video/x-mng,image/
png,image/jpeg,image/gif;q=0.2,??
```

12.3.2 TCP 套接口 Client/Server 程序实例

本小节将讲述一个面向连接的套接口通信实例。面向连接的网络通信，包括客户端和服务器端两个部分的程序。服务器实现监听功能，如果有客户端请求连接，则接受这个连接。客户端实现请求连接的功能，可以发送请求到服务器。当 TCP 连接建立完后，客户端和服务器端便可以进行通信了，客户端可以向服务器发送信息，服务器接收客户端的信息并进行应答，客户端接收应答，循环下一次通信。

下面给出基于 TCP 套接口的 Client/Server 程序，并详细给出程序的运行结果。

1. 服务器端程序

所谓服务器程序，指的是在网络通信时这个程序始终处于等待状态，可以接受用户的连接请求，并且对用户发送的信息进行处理，程序 12.6 是面向连接的套接口通信服务器端程序。程序 12.6 实现的功能是接受用户从终端输入的字符串数据，并将数据保持在发送缓冲区内，当 TCP 连接建立以后，接收请求连接的客户机的 IP 地址信息，然后将用户输入的数据信息(存放在缓冲区)发送给客户机。源代码如 server_tcp.c 所示。

【程序 12.6】基于 TCP 协议的服务器端程序：server_tcp.c。

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <unistd.h>
#define MAXSIZE 1024        /*定义数据缓冲区大小*/

int main(int argc, char *argv[])
{
int sockfd,new_fd;
struct sockaddr_in server_addr;     /*定义服务器端套接口数据结构 server_addr */
struct sockaddr_in client_addr;     /*定义客户端套接口数据结构 client_addr */
int sin_size,portnumber;
char buf[MAXSIZE];        /*发送数据缓冲区*/
if(argc!=2)
{
fprintf(stderr,"Usage:%s portnumber\a\n",argv[0]);
exit(1);
}
if((portnumber=atoi(argv[1]))<0)
{   /*获得命令行的第二个参数——端口号，atoi()把字符串转换成整型数*/
fprintf(stderr,"Usage:%s portnumber\a\n",argv[0]);
exit(1);
}
if((sockfd=socket(AF_INET,SOCK_STREAM,0))==-1)
/*服务器端开始建立 socket 描述符*/
{
fprintf(stderr,"Socket error:%s\n\a",strerror(errno));
exit(1);
}
/*服务器端填充 sockaddr 结构*/
bzero(&server_addr,sizeof(struct sockaddr_in)); /*先将套接口地址数据结构清零*/
server_addr.sin_family=AF_INET;
server_addr.sin_addr.s_addr=htonl(INADDR_ANY);
server_addr.sin_port=htons(portnumber);
if(bind(sockfd,(struct sockaddr *)(&server_addr),sizeof(struct sockaddr))==-1)
```

```
                                        /*调用 bind 函数绑定端口*/
    {
    fprintf(stderr,"Bind error:%s\n\a",strerror(errno));
    exit(1);
    }
    if(listen(sockfd,5)==-1)
    /*端口绑定成功，监听 sockfd 描述符，同时处理的最大连接请求数为 5 */
    {
    fprintf(stderr,"Listen error:%s\n\a",strerror(errno));
    exit(1);
    }
    while(1)        /*服务器阻塞，等待接收连接请求，直到客户程序发送连接请求*/
    {
    sin_size=sizeof(struct sockaddr_in);
    if((new_fd=accept(sockfd,(struct sockaddr *)(&client_addr),&sin_size))==-1)
    /*调用 accept 接受一个连接请求*/
    {
    fprintf(stderr,"Accept error:%s\n\a",strerror(errno));
    exit(1);
    }
    fprintf(stderr,"Server get connection from %s\n",inet_ntoa(client_addr.sin_addr));
    /*TCP 连接已建立，打印申请连接的客户机的 IP 地址*/
    printf("Connected successful, please input the message[<1024 bytes]:\n");
            /*提示用户输入将要发送的数据，长度小于缓冲区的长度，即 1024 字节*/
    if(fgets(buf, sizeof(buf), stdin) != buf)
    {  /*从终端输入的数据存放在 buf 缓冲区*/
            printf("fgets error!\n");
            exit(1);
    }
    if(write(new_fd,buf,strlen(buf))==-1)     /*调用 write 发送数据*/
    {
    fprintf(stderr,"Write Error:%s\n",strerror(errno));
    exit(1);
    }
    close(new_fd);   /*本次通信已结束，关闭客户端的套接口，并循环下一次等待*/
    }
    close(sockfd);   /*服务器进程结束，关闭服务器端套接口*/
    exit(0);
    }
```

2. 客户端程序

客户端指的是在网络通信时主动向服务器发送连接请求，主动发送信息的程序。程序 12.7 是面向连接的套接口通信的客户端程序。这个程序的主要内容是向服务器申请连接，并且接收服务器发来的数据，最后打印出接收到的数据信息。源代码如 client _tcp.c 所示。

【程序 12.7】基于 TCP 协议的客户端程序：client_tcp.c。

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
```

```
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

int main(int argc, char *argv[])
{
int sockfd;
char buffer[1024];
struct sockaddr_in server_addr; /*定义服务器端套接口数据结构 server_addr */
struct hostent *host;
int portnumber,nbytes;
if(argc!=3)
{
fprintf(stderr,"Usage:%s hostname portnumber\a\n",argv[0]);
exit(1);
}
if((host=gethostbyname(argv[1]))==NULL)
{   /*获得命令行的第二个参数——主机名*/
fprintf(stderr,"Gethostname error\n");
exit(1);
}
if((portnumber=atoi(argv[2]))<0)
{   /*获得命令行的第三个参数——端口号，atoi()把字符串转换成整型数*/
fprintf(stderr,"Usage:%s hostname portnumber\a\n",argv[0]);
exit(1);
}
/*客户程序开始建立 sockfd 描述符*/
if((sockfd=socket(AF_INET,SOCK_STREAM,0))==-1)
{
fprintf(stderr,"Socket Error:%s\a\n",strerror(errno));
exit(1);
}
/*客户程序填充服务端的资料*/
bzero(&server_addr,sizeof(server_addr));
server_addr.sin_family=AF_INET;
server_addr.sin_port=htons(portnumber);
server_addr.sin_addr=*((struct in_addr *)host->h_addr);
/*客户程序发起连接请求*/
if(connect(sockfd,(struct sockaddr *)(&server_addr),sizeof(struct sockaddr))==-1)
{
fprintf(stderr,"Connect Error:%s\a\n",strerror(errno));
exit(1);
}
/*连接成功，调用 read 读取服务器发送来的书籍*/
if((nbytes=read(sockfd,buffer,1024))==-1)
{
```

```
        fprintf(stderr,"Read Error:%s\n",strerror(errno));
        exit(1);
        }
        buffer[nbytes]='\0';
        printf("I have received:%s\n",buffer);      /*输出接收到的数据*/
        close(sockfd);      /*结束通信*/
        exit(0);
        }
```

此时打开两个 shell 终端，分别使用 gcc 编译 server_tcp.c 和 client _tcp.c，并生成相应的可执行文件：

```
# gcc -o server_tcp server_tcp.c
# gcc -o client_tcp client_tcp.c
```

然后在一个 shell 下运行服务器端程序 server_tcp，命令如下：

```
# ./server_tcp 5678        /*命令行的第二个参数为端口号(5678)*/
```

此时在另一个 shell 下运行客户端程序 client _tcp，命令如下：

```
# ./client_tcp localhost 5678    /*命令行参数包括将要连接的服务器主机名和端口号，这里表示连接本机上的 5678 号端口*/
```

此时观察服务器端 shell 的输出为：

```
Server get connection from 192.168.1.88
Connected successful, please input the message[<1024 bytes]:
```

输出表明服务器已经接受了客户机的连接请求，并已建立好连接，接收到客户机传来的信息——客户机的 IP 地址，并提示用户输入将要发送给客户机的数据(小于 1024 字节)，此时我们从终端输入：

```
Hello! I like Linux C programs! ↙(↙表示回车)
```

这时再来查看客户端 shell 中的输出：

```
I have received: Hello! I like Linux C programs!
```

说明服务器和客户机已经真正建立了通信连接，并可以正确收发信息了。

最后，总的来说，网络程序是由两个部分组成的——客户端和服务器端，它们的建立步骤一般是：

- 服务器端：socket-->bind-->listen-->accept
- 客户端：socket-->connect

通过这个实例的学习，读者可以看到本小节中所讲述的套接口连接的综合操作。这个实例是网络传输的最基本的原理和形式，用这种方法还可以实现不同计算机之间的文本、文件等类型数据的传输。

12.4 UDP 套接口编程

不同于 TCP 协议，UDP 是一个无连接、不可靠服务的运输层协议，它不对数据进行确认、出错重传和排序等可靠性处理，但是它却具有代码小、实现简单、速度快和系统开销小等优点。对应某些应用，使用 UDP 将带来更高的效率，如域名服务系统 DNS、网络文件系统 NFS 等。

12.4.1 UDP 套接口通信工作流程

TCP 和 UDP 的最大区别是 TCP 是面向连接的，而 UDP 是无连接的。基于 UDP 传输协议的服务器与客户机间的通信工作流程可以用如图 12.9 所示的过程来描述。

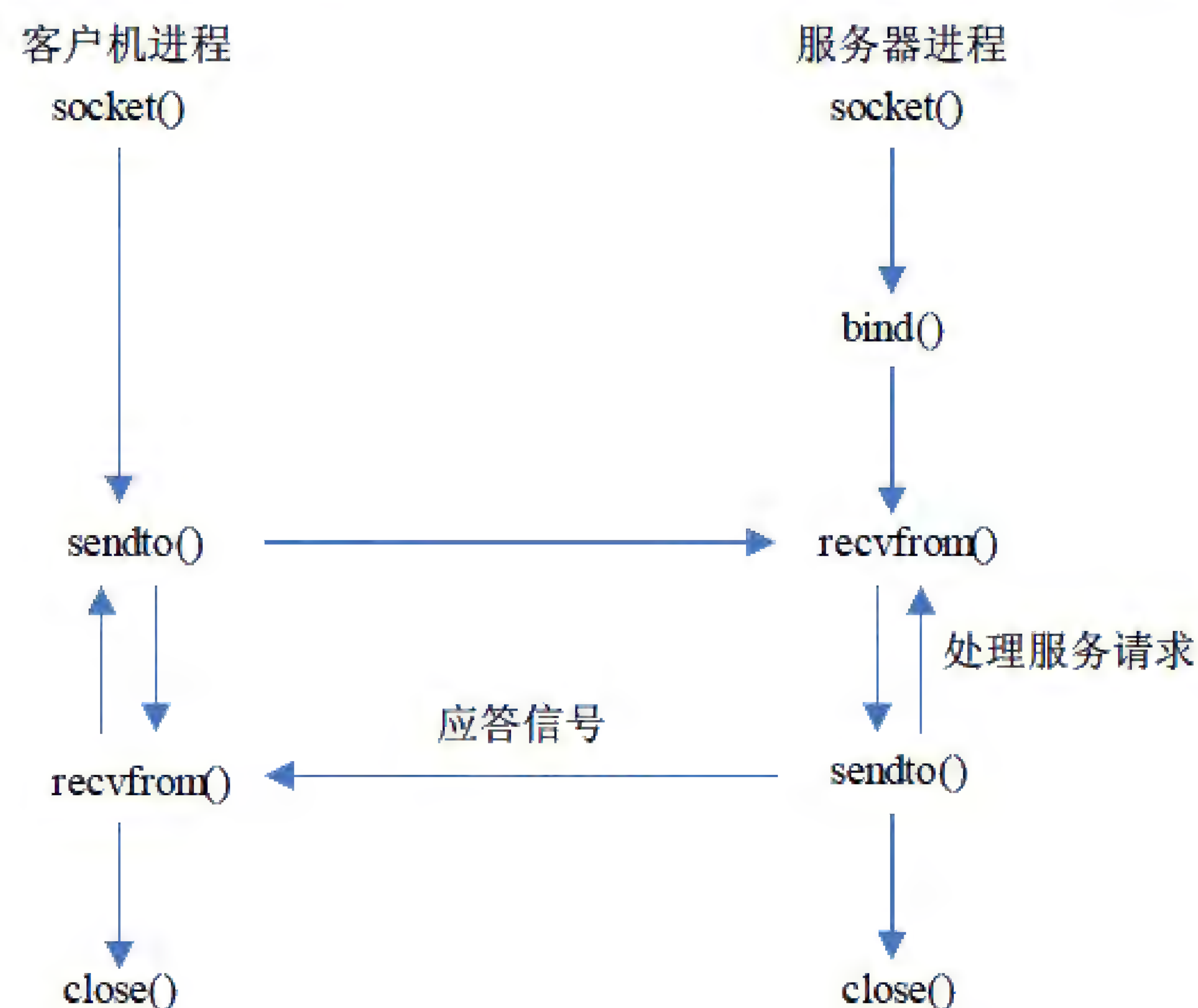


图 12.9 UDP 套接口通信工作流程

将上图与图 12.8 相比较，它们的主要区别在于：使用 TCP 套接口必须先建立连接(如客户进程的 connect()，服务器进程的 listen()和 accept())，而 UDP 套接口不需要预先建立连接，它在调用 socket()生成一个套接口后，在服务器端调用 bind()绑定一个端口，然后服务器进程挂起于 recvfrom()调用，等待并接收网络中某一客户机的数据请求，而客户端调用 sendto()发送数据请求，同样也挂起于 recvfrom()调用，等待并接收服务器的应答信号。当数据传送完毕后，UDP 套接口中的客户端调用 close()释放通信链路，但不再发送“断开连接通知”(见图 12.8)信息来通知服务器端释放通信链路。

以上便是基于 UDP 传输协议的套接口通信的工作流程，其中的一些函数调用与 TCP 的套接口相同，在 12.3.1 小节中已经详细介绍过了。下面介绍基于 UDP 协议的数据发送和数据接收函数。

1. 数据发送函数

基于 UDP 套接口的数据发送函数为 sendto，它的作用相当于 TCP 中的 send 函数或 write 函数。sendto 的函数原型如下：

```
#include <sys/socket.h>
int sendto (int sockfd, const void *msg, int len, unsigned int flags,
struct sockaddr *toaddr, int *addrlen);
```

返回：若成功则返回实际发送的字节数，若出错则返回-1。

参数 sockfd 为套接口的描述符，msg 为指向数据发送缓冲区的指针，len 表示将要发送的字节数，flags 一般设置为 0，toaddr 为指向数据发送的套接口地址数据结构的指针，addrlen 指向套接口数据结构的长度。

2. 数据接收函数

基于 UDP 套接口的数据接收函数为 recvfrom，它的作用相当于 TCP 中的 recv 函数或 read 函数。recvfrom 的函数原型如下：

```
#include <sys/socket.h>
int recvfrom (int sockfd, void *buf, int len, unsigned int flags,
struct sockaddr *fromaddr, int *addrlen);
```

返回：若成功则返回实际接收的字节数，若出错则返回-1。

参数 buf 指向数据接收缓冲区的指针，fromaddr 为指向数据接收的套接口地址结构的指针，其他参数的含义与 sendto 相同。

12.4.2 UDP 套接口 Client/Server 程序实例

本小节将讲述一个无连接的套接口通信实例。无连接的网络通信，同样包括客户端和服务器端两个部分的程序。在下面的实例中，客户端和服务器端可以互相发送信息，并互相接收对方的信息，并且这两个程序完全可以在两个计算机之间实现字符串传输，经测试，在广域网中是可以正常运行的。

下面给出基于 UDP 套接口的 Client/Server 程序，并详细给出程序的运行结果。

1. 服务器端程序

程序 12.8 是基于 UDP 套接口的服务器端程序，程序中定义了一个固定的端口号 8888。程序的功能是接收用户终端输入的字符数据，发送给客户机，还可以接收客户机发来的数据，并打印出来。源代码如 server_udp.c 所示。

【程序 12.8】基于 UDP 协议的服务器端程序：server_udp.c。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <errno.h>

#define SERVER_PORT 8888        /*定义端口号*/
#define MAX_MSG_SIZE 1024       /*服务器发送和接收数据缓冲区的大小*/

int main(void)
{
int sockfd,addrlen,n;
```

```
struct sockaddr_in addr;     /*定义服务器端套接口地址数据结构 addr */
char msg[MAX_MSG_SIZE];
sockfd=socket(AF_INET,SOCK_DGRAM,0); /*服务器端开始建立 socket 描述符*/
if(sockfd<0)
{
fprintf(stderr,"Socket Error:%s\n",strerror(errno));
exit(1);
}
addrlen=sizeof(struct sockaddr_in);
bzero(&addr,addrlen);
/*服务器端填充 sockaddr 结构*/
addr.sin_family=AF_INET;
addr.sin_addr.s_addr=htonl(INADDR_ANY);
addr.sin_port=htons(SERVER_PORT);
if(bind(sockfd,(struct sockaddr *)(&addr),addrlen)<0)    /*调用 bind 函数绑定端口*/
{
fprintf(stderr,"Bind Error:%s\n",strerror(errno));
exit(1);
}
while(1)
{   /*从网络中读取数据，并打印出接收到的数据*/
bzero(msg,MAX_MSG_SIZE);    /*接收数据之前先将缓冲区清零*/
n=recvfrom(sockfd,msg,sizeof(msg),0,(struct sockaddr*)(&addr),&addrlen);
fprintf(stdout,"Receive message from client: %s",msg);
/*从终端读入用户输入的数据，发送到网络中去*/
fgets(msg,MAX_MSG_SIZE,stdin);
printf("Server endpoint input message:%s",msg);
sendto(sockfd,msg,n,0,(struct sockaddr*)(&addr),addrlen);
}
close(sockfd); /*通信结束，关闭套接口*/
exit(0);
}
```

2. 客户端程序

程序 12.9 是基于 UDP 套接口的客户端程序。客户机将要与之进行的服务器的 IP 地址和端口号由命令行的方式传递给程序，程序 12.9 的主要功能与程序 12.8 类似，即接收用户终端输入的字符数据，发送给服务器，还可以接收服务器发来的数据，并打印出来。源代码如 client_udp.c 所示。

【程序 12.9】基于 UDP 协议的客户端程序：client_udp.c。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <errno.h>
#include <stdio.h>
#include <unistd.h>

#define MAX_BUF_SIZE 1024     /*客户端发送和接收数据缓冲区的大小*/
```

```
int main(int argc,char **argv)
{
int sockfd,port,addrlen,n;
char buffer[MAX_BUF_SIZE];
struct sockaddr_in addr;
if(argc!=3)
{
fprintf(stderr,"Usage:%s server_ip server_port\n",argv[0]);
exit(1);
}
if((port=atoi(argv[2]))<0)
{   /*命令行的第三个参数为数据将要发送到的服务器端口号*/
fprintf(stderr,"Usage:%s server_ip server_port\n",argv[0]);
exit(1);
}
sockfd=socket(AF_INET,SOCK_DGRAM,0);   /*建立客户端 socket*/
if(sockfd<0)
{
fprintf(stderr,"Socket Error:%s\n",strerror(errno));
exit(1);
}
addrlen=sizeof(struct sockaddr_in);
bzero(&addr,addrlen);
/*客户端填充 sockaddr 结构*/
addr.sin_family=AF_INET;
addr.sin_port=htons(port);
if(inet_aton(argv[1],&addr.sin_addr)<0)
{  /*命令行的第三个参数为数据将要发送到的服务器 IP 地址*/
fprintf(stderr,"Ip error:%s\n",strerror(errno));
exit(1);
}
while(1)
{  /*从键盘读入，发送到服务器端*/
bzero(buffer,MAX_BUF_SIZE); /*接收数据之前先将缓冲区清零*/
fgets(buffer,MAX_BUF_SIZE,stdin);
sendto(sockfd,buffer,strlen(buffer),0,(struct sockaddr *)(&addr),addrlen);
printf("Client endpoint input message: %s",buffer);
/*从网络中读取数据，并打印出接收到的数据*/
n=recvfrom(sockfd,buffer,strlen(buffer),0,(struct sockaddr *)(&addr),&addrlen);
fprintf(stdout,"Receive message from server: %s",buffer);
}
close(sockfd);   /*通信结束，关闭套接口*/
exit(0);
}
```

为了测试这两个程序，我们同样打开两个 shell 终端，分别使用 gcc 编译 server_udp.c 和 client_udp.c，并生成相应的可执行文件：

```
# gcc -o server_udp server_udp.c
# gcc -o client_udp client_udp.c
```

然后在一个 shell 下运行服务器端程序 server_udp，命令如下：

```
# ./server_udp
```

此时在另一个 shell 下运行客户端程序 client _udp，命令如下：

```
# ./client_udp localhost 8888    /*命令行参数包括将要连接的服务器主机名和端口号，这里表示连接本机上的 8888 号端口*/
```

接着在客户端的 shell 终端下输入字符数据，键入“回车”键后输出如下结果：

```
Hello, I like Linux C programs!↙(↙表示回车)
Client endpoint input message: Hello, I like Linux C programs!
```

此时观察服务器端 shell 的输出为：

```
Receive message from client: Hello, I like Linux C programs!
```

接着在服务器端的 shell 下输入字符数据，键入“回车”键后输出如下结果：

```
I am doing Linux C programs!↙(↙表示回车)
Server endpoint input message: I am doing Linux C programs!
```

此时观察客户端 shell 的输出为：

```
Receive message from server: I am doing Linux C programs!
```

可以看到，客户端和服务器已经可以正常通信了。

无连接的套接口通信是一种简单的通信方式，编程的重点是信息的发送和接收，通过本小节的实例，可以实现两个计算机之间的文字信息传输，读者可以在这两个程序的基础上编写出文本模式下的简单聊天软件。(事实上，在本书的第 3 部分实践篇中会给出一个基于 GTK+图形开发库的图形化聊天软件的开发实例。)

12.5 原始套接口编程

前面两节我们介绍了两种套接口(SOCK_STREAM 和 SOCK_DRAGM)的编程，在这一节中将向读者讲解另外一种套接口——原始套接口(SOCK_RAW)。应用原始套接口，可以编写出 TCP 和 UDP 套接口不能够实现的功能。需要注意的是，原始套接口只能由有 root 权限的用户创建。相比于 TCP 和 UDP 套接口，原始套接口具有以下功能：

- 使用原始套接口可以读/写 ICMP(互联网控制消息协议)及 ICMPv6 分组，如 ping 就使用原始套接口发送 ICMP 应答请求。
- 使用原始套接口可以读/写特殊的 IP 数据包，内核不处理这些数据包的 IP 协议字段，而出错的诊断将依靠协议字段的意义。
- 利用原始套接口通过设置 IP_HDRINCL 套接口选项可以构造自己的 IP 头部。

基于原始套接口编程相关系统的调用与TCP和UDP套接口相同，比如函数 socket()、bind()、

connect()等都能使用，下面简单介绍原始套接口的创建之后，给出一个具体的实例来说明它的使用方法。

12.5.1 原始套接口的创建

sockfd()函数也可以用来创建一个原始套接口，创建的形式如下：

```
int sockfd;
sockfd = sockfd (AF_INET, SOCK_RAW, protocol);
```

AF_INET 表示使用的是 IPv4 协议族，SOCK_RAW 表明是原始套接口，protocol 是协议名，回忆在 TCP 和 UDP 套接口的创建时，参数 protocol 取值是为 0 的，在这里(创建原始套接口)，它的取值可以是 IPPROTO_ICMP、IPPROTO_TCP、IPPROTO_UDP 等，不同的协议类型创建不同类型的原始套接口。更详细的信息读者可以查看<netinet/in.h>的源代码。

创建原始套接口以后，可以通过它向网络中发送自己定义的 IP 数据包，为了防止非法用户破坏网络，规定只有超级用户 root 才具有创建原始套接口的权限。下面以一个实例来说明原始套接字的创建和使用。

12.5.2 原始套接口程序实例

这里举一个较常用的 ping 程序的例子。在这个程序中，使用 IPPROTO_ICMP 参数选项创建一个 ICMP 原始套接口，并利用这个套接口收发数据，该程序实现了常见的 ping 命令的功能。

1. ping 命令简介

ping 命令是用来查看本地主机与网络中另一个主机系统的网络连接是否正常的一个工具。ping 命令的工作原理是由本地主机向网络上的另一个主机系统发送 ICMP 报文，如果指定系统得到了报文，它将把报文一模一样地传回给发送者，这有点像潜水艇声纳系统中使用的发声装置。在 TCP/IP 体系结构中，ping 是应用层直接使用网络层 ICMP 的例子，它没有通过运输层的 TCP 或 UDP。

ping 命令是 Linux 用户很熟悉的一个命令了。例如，笔者使用 ping 命令查看本地主机与局域网中某一主机系统(192.168.1.103)的网络连接情况，可以在 Linux shell 终端上执行 ping 192.168.1.103 命令，将会看到以下结果：

```
#ping 192.168.1.103
PING 192.168.1.103 (192.168.1.103) 56(84) bytes of data.
64 bytes from 192.168.1.103 (192.168.1.103): icmp_seq=1 ttl=64 time=0.029ms
64 bytes from 192.168.1.103 (192.168.1.103): icmp_seq=2 ttl=64 time=0.047ms
64 bytes from 192.168.1.103 (192.168.1.103): icmp_seq=3 ttl=64 time=0.053ms
64 bytes from 192.168.1.103 (192.168.1.103): icmp_seq=4 ttl=64 time=0.031ms

--- localhost.localdomain ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss, time 2997ms
rtt min/avg/max/mdev = 0.029/0.040/0.053/0.010 ms
```

由上面的执行结果可以看到，ping 命令执行后显示出被测试系统主机名和相应 IP 地址、返回给当前主机的 ICMP 报文顺序号、ttl 生存时间和往返时间 rtt(单位是毫秒，即千分之一秒)。

要写一个模拟 ping 命令，这些信息有启示作用。

2. ICMP 协议简介

要真正了解 ping 命令实现原理，就要了解 ping 命令所使用到的 TCP/IP 协议。ICMP(Internet Control Message，网际控制报文协议)是为网关和目标主机而提供的一种差错控制机制，使它们在遇到差错时能把错误报告给报文源发送方。ICMP 协议是 IP 层的一个协议，但是由于差错报告在发送给报文源发送方时可能也要经过若干子网，因此牵涉到路由选择等问题，所以 ICMP 报文需通过 IP 协议来发送。ICMP 数据报的数据发送前需要两级封装：首先添加 ICMP 报头形成 ICMP 报文，再添加 IP 报头形成 IP 数据报，封装过程如图 12.10 所示。

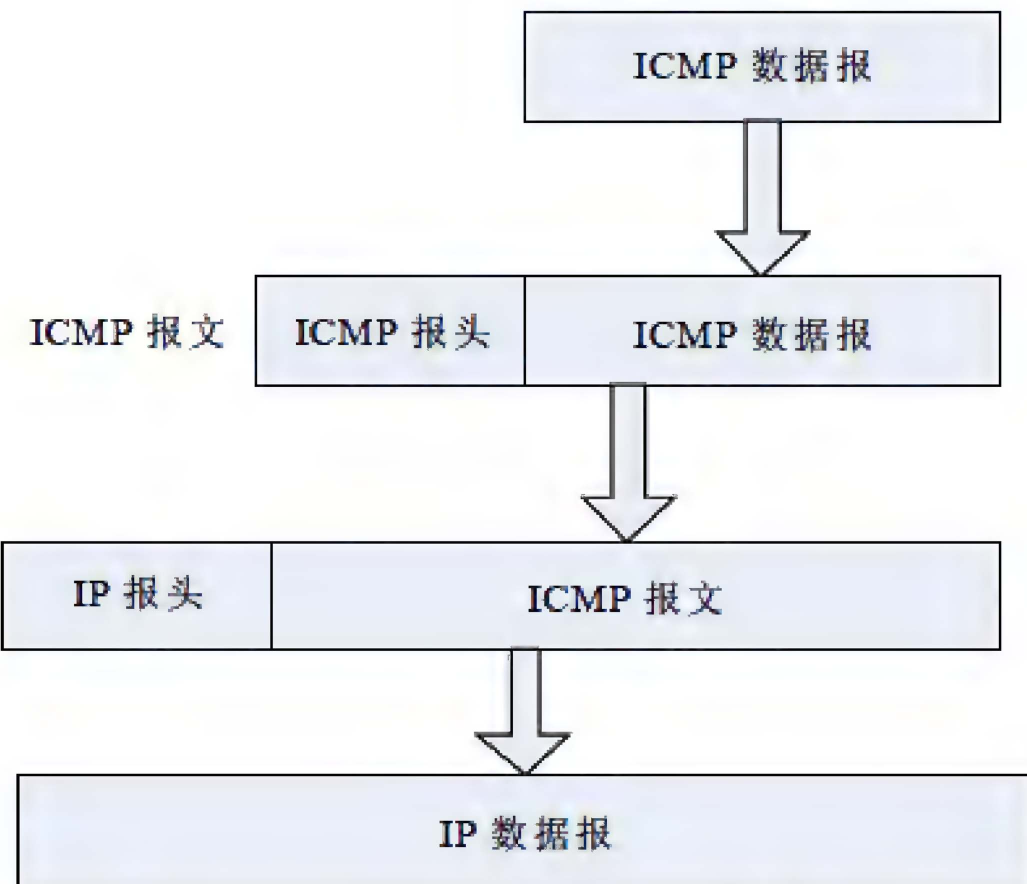


图 12.10　ICMP 数据报封装过程

3. IP 报头格式

由于 IP 层协议是一种点对点的协议，而非端对端的协议，它提供无连接的数据报服务，没有建立端口连接的概念，因此很少使用 bind()和 connect()函数，若有使用也只是用于设置 IP 地址。发送数据使用 sendto()函数，接收数据使用 recvfrom()函数。IP 报头格式如图 12.11 所示。

比特 0	4	8	16		31
版本号 VER	IP 报头长度	服务类型 TOS	数据报总长度 TL		
报文标识 ID			标志	分段偏移量 FO	
生存时间 TTL		协议号 PORT	报头检验和		
源地址					
目的地址					
任选项和填充位					

图 12.11　IP 报头格式

在 Linux 中，IP 报头格式数据结构的定义如下(取自<netinet/ip.h>)：

```
struct ip
{
#if __BYTE_ORDER==__LITTLE_ENDIAN
unsigned int ip_hl:4;      /* header length */
unsigned int ip_v:4;       /* version */
```

```
#endif
#if __BYTE_ORDER==__BIG_ENDIAN
unsigned int ip_v:4;      /* version */
unsigned int ip_hl:4;   /* header length */
#endif
u_int8_t ip_tos;        /* type of service */
u_short ip_len;         /* total length */
u_short ip_id;          /* identification */
u_short ip_off;         /* fragment offset field */
#define IP_RF 0x8000 /* reserved fragment flag */
#define IP_DF 0x4000 /* dont fragment flag */
#define IP_MF 0x2000 /* more fragments flag */
#define IP_OFFMASK 0x1fff /* mask for fragmenting bits */
u_int8_t ip_ttl;        /* time to live */
u_int8_t ip_p;          /* protocol */
u_short ip_sum;     /* checksum */
struct in_addr ip_src, ip_dst;     /* source and dest address */
};
```

其中，ping 程序只使用以下数据：

- IP 报头长度 IHL(Internet Header Length)：以 4 字节为一个单位来记录 IP 报头的长度，是上述 IP 数据结构的 ip_hl 变量。
- 生存时间 TTL(Time To Live)：以秒为单位，指出 IP 数据报能在网络上停留的最长时间，其值由发送方设定，并在经过路由的每一个节点时减一，当该值为 0 时，数据报将被丢弃，是上述 IP 数据结构的 ip_ttl 变量。

4. ICMP 报头格式

ICMP 报文分为两种，一种是错误报告报文；另一种是查询报文。ICMP 差错报告报文共有 5 种：

- 终点不可达。
- 源站抑制。
- 时间超过。
- 参数问题。
- 改变路由(重定向)。

ICMP 的查询报文又分为 4 种：

- 回送请求和回答报文。
- 时间戳请求和回答报文。
- 掩码地址请求和回答报文。
- 路由器询问和通告报文。

ICMP 的报头共有 8 字节，前 4 个字节采用统一的格式，共有 3 个字段，即类型、代码和检验和，长度分别为 8 位、8 位和 16 位。接着的 4 个字节的内容与 ICMP 的类型有关。

ping 命令只使用众多 ICMP 报文中的两种：请求回送(ICMP_ECHO)和请求回应(ICMP_ECHOREPLY)。在 Linux 中的定义如下(取自<netinet/ip_icmp.h>)：

```
#define ICMP_ECHO 0
```

```
#define ICMP_ECHOREPLY 8
```

这两种 ICMP 类型报头格式如图 12.12 所示。



图 12.12 ICMP 报头格式

在 Linux 中，ICMP 数据结构的定义如下(取自<netinet/ip_icmp.h>)：

```
struct icmp
{
  u_int8_t   icmp_type;      /* type of message, see below */
  u_int8_t   icmp_code;      /* type sub code */
  u_int16_t icmp_cksum;     /* ones complement checksum of struct */
  union
  {
    u_char ih_pptr;          /* ICMP_PARAMPROB */
    struct in_addr ih_gwaddr;     /* gateway address */
    struct ih_idseq          /* echo datagram */
    {
      u_int16_t icd_id;
      u_int16_t icd_seq;
    } ih_idseq;
    u_int32_t ih_void;

    /* ICMP_UNREACH_NEEDFRAG -- Path MTU Discovery (RFC1191) */
    struct ih_pmtu
    {
      u_int16_t ipm_void;
      u_int16_t ipm_nextmtu;
    } ih_pmtu;

    struct ih_rtradv
    {
      u_int8_t irt_num_addrs;
      u_int8_t irt_wpa;
      u_int16_t irt_lifetime;
    } ih_rtradv;
  } icmp_hun;
#define    icmp_pptr  icmp_hun.ih_pptr
#define    icmp_gwaddr     icmp_hun.ih_gwaddr
#define    icmp_id         icmp_hun.ih_idseq.icd_id
#define    icmp_seq   icmp_hun.ih_idseq.icd_seq
#define    icmp_void icmp_hun.ih_void
#define    icmp_pmvoid     icmp_hun.ih_pmtu.ipm_void
#define    icmp_nextmtu    icmp_hun.ih_pmtu.ipm_nextmtu
#define    icmp_num_addrsicmp_hun.ih_rtradv.irt_num_addrs
```

```
#define    icmp_wpa icmp_hun.ih_rtradv.irt_wpa
#define    icmp_lifetime    icmp_hun.ih_rtradv.irt_lifetime
  union
  {
    struct
    {
      u_int32_t its_otime;
      u_int32_t its_rtime;
      u_int32_t its_ttime;
    } id_ts;
    struct
    {
      struct ip idi_ip;
      /* options and then 64 bits of data */
    } id_ip;
    struct icmp_ra_addr id_radv;
    u_int32_t    id_mask;
    u_int8_t     id_data[1];
  } icmp_dun;
#define    icmp_otime    icmp_dun.id_ts.its_otime
#define    icmp_rtimeicmp_dun.id_ts.its_rtime
#define    icmp_ttimeicmp_dun.id_ts.its_ttime
#define    icmp_ip       icmp_dun.id_ip.idi_ip
#define    icmp_radv icmp_dun.id_radv
#define    icmp_maskicmp_dun.id_mask
#define    icmp_data icmp_dun.id_data
};
```

使用宏定义命令表达更简洁，其中 ICMP 报头为 8 字节，数据报长度最大为 64KB 字节。

下面介绍几个在本小节的实例程序中将要用到的概念：

- 校验和算法：这一算法称为网际校验和算法，把被校验的数据进行 16 位累加，然后取反码，若数据字节长度为奇数，则数据尾部补上一个字节的 0 以凑成偶数。此算法适用于 IPv4、ICMPv4、IGMPv4、ICMPv6、UDP 和 TCP 校验和，更详细的信息请参考 RFC1071，校验和字段为上述 ICMP 数据结构的 icmp_cksum 变量。
- 标识符：用于唯一标识 ICMP 报文，为上述 ICMP 数据结构的 icmp_id 宏所指的变量。
- 顺序号：ping 命令的 icmp_seq 便由这里读出，代表 ICMP 报文的发送顺序，为上述 ICMP 数据结构的 icmp_seq 宏所指的变量。

在本实例中，ping 命令需要显示的信息，包括 icmp_seq 和 ttl 都已有实现的办法，但还缺少往返时间 rtt 的算法实现。为了实现这一功能，可利用 ICMP 数据报携带一个时间戳。使用下列函数生成一个时间戳：

```
#include <sys/time.h>
#include <unistd.h>
int gettimeofday(struct timeval *tv,void *tz);
```

返回：若成功，则返回 0；若失败，则返回-1。错误代码存于 errno。

gettimeofday()函数会把目前的时间由 tv 所指向的结构体 timeval 返回，当地时区的信息则

放到 tz 所指向的结构体 timezone 中。

其中 timeval 结构的定义如下(细心的读者会发现，我们在本书的 9.2.2 小节中已经介绍过这个结构)：

```
struct timeval
{
    long tv_sec;            /*时间的秒数部分*/
    long tv_usec;           /*时间的微秒(1/1000000)部分*/
};
```

timezone 结构的定义如下：

```
struct timezone
{
int tz_minuteswest;     /*和 Greenwich 时间差了多少分钟*/
int tz_dsttime;         /*日光节约时间的状态*/
};
```

上述两个结构都定义在/usr/include/sys/time.h 文件中。tz_dsttime 所代表的状态如下：

```
DST_NONE /*不使用*/
DST_USA /*美国*/
DST_AUST /*澳洲*/
DST_WET /*西欧*/
DST_MET /*中欧*/
DST_EET /*东欧*/
DST_CAN /*加拿大*/
DST_GB /*大不列颠*/
DST_RUM /*罗马尼亚*/
DST_TUR /*土耳其*/
DST_AUSTALT /*澳洲(1986 年以后)*/
```

timeval 结构中，tv_sec 为秒数，tv_usec 为微秒数。在发送和接收报文时由 gettimeofday 分别生成两个 timeval 结构，两者之差即为往返时间，即 ICMP 报文发送与接收的时间差，而 timeval 结构由 ICMP 数据报携带，tz 指针表示时区，一般都不使用，赋 NULL 值。

最后，系统自带的 ping 命令当它接送完所有 ICMP 报文后，会对所有发送和所有接收的 ICMP 报文进行统计，从而计算 ICMP 报文丢失的比率。为达此目的，定义两个全局变量：接收计数器和发送计数器，用于记录 ICMP 报文接收和发送数目。丢失数目=发送总数 - 接收总数，丢失比率=丢失数目/发送总数。

5. ping 程序代码

下面给出完整的 ping 程序代码，如 myping.c 所示。

【程序 12.10】自己编写的 ping 程序代码：myping.c。

```
#include <stdio.h>
#include <signal.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
```

```
#include <unistd.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netdb.h>
#include <setjmp.h>
#include <errno.h>

#define PACKET_SIZE 4096
#define MAX_WAIT_TIME 5
#define MAX_NO_PACKETS 3

char sendpacket [PACKET_SIZE];
char recvpacket [PACKET_SIZE];
int sockfd,datalen=56;
int nsend=0,nreceived=0;
struct sockaddr_in dest_addr;
pid_t pid;
struct sockaddr_in from;
struct timeval tvrecv;

void statistics(int signo);
unsigned short cal_chksum(unsigned short *addr,int len);
int pack(int pack_no);
void send_packet(void);
void recv_packet(void);
int unpack(char *buf,int len);
void tv_sub(struct timeval *out,struct timeval *in);

void statistics(int signo)
{
printf("\n--------------------PING statistics-------------------\n");
printf("%d packets transmitted, %d received , %%%d lost\n",nsend,nreceived,
(nsend-nreceived)/nsend*100);
close(sockfd);
exit(1);
}

/*校验和算法*/
unsigned short cal_chksum(unsigned short *addr,int len)
{
int nleft=len;
int sum=0;
unsigned short *w=addr;
unsigned short answer=0;
/*把 ICMP 报头二进制数据以 2 字节为单位累加起来*/
while(nleft>1)
{
sum+=*w++;
nleft-=2;
```

```
}
/*若 ICMP 报头为奇数个字节，会剩下最后一字节。把最后一个字节视为一个 2 字节数据的高字节，这个
2 字节数据的低字节为 0，继续累加*/
if( nleft==1)
{
*(unsigned char *)(&answer)=*(unsigned char *)w;
sum+=answer;
}
sum=(sum>>16)+(sum&0xffff);
sum+=(sum>>16);
answer=~sum;
return answer;
}

/*设置 ICMP 报头*/
int pack(int pack_no)
{
int i,packsize;
struct icmp *icmp;
struct timeval *tval;

icmp=(struct icmp*)sendpacket;
icmp->icmp_type=ICMP_ECHO;
icmp->icmp_code=0;
icmp->icmp_cksum=0;
icmp->icmp_seq=pack_no;
icmp->icmp_id=pid;
packsize=8+datalen;
tval= (struct timeval *)icmp->icmp_data;
gettimeofday(tval,NULL); /*记录发送时间*/
icmp->icmp_cksum=cal_chksum( (unsigned short *)icmp,packsize); /*校验算法*/
return packsize;
}

/*发送 3 个 ICMP 报文*/
void send_packet()
{
int packetsize;
while(nsend<MAX_NO_PACKETS)
{
nsend++;
packetsize=pack(nsend); /*设置 ICMP 报头*/
if( sendto(sockfd,sendpacket,packetsize,0,
(struct sockaddr *)&dest_addr,sizeof(dest_addr) )<0 )
{
perror("sendto error");
continue;
}
sleep(1); /*每隔一秒发送一个 ICMP 报文*/
}
```

```
}

/*接收所有 ICMP 报文*/
void recv_packet()
{
int n,fromlen;
extern int errno;

signal(SIGALRM,statistics);
fromlen=sizeof(from);
while(nreceived<nsend)
{
alarm(MAX_WAIT_TIME);
if( (n=recvfrom(sockfd,recvpacket,sizeof(recvpacket),0,
(struct sockaddr *)&from,&fromlen)) <0)
{
if(errno==EINTR)
continue;
perror("recvfrom error");
continue;
}
gettimeofday(&tvrecv,NULL); /*记录接收时间*/
if(unpack(recvpacket,n)==-1)continue;
nreceived++;
}
}

/*剥去 ICMP 报头*/
int unpack(char *buf,int len)
{
int i,iphdrlen;
struct ip *ip;
struct icmp *icmp;
struct timeval *tvsend;
double rtt;

ip=(struct ip *)buf;
iphdrlen=ip->ip_hl<<2; /*求 IP 报头长度，即 IP 报头的长度标志乘 4*/
icmp=(struct icmp *)(buf+iphdrlen); /*越过 IP 报头，指向 ICMP 报头*/
len-=iphdrlen; /*ICMP 报头及 ICMP 数据报的总长度*/
if( len<8) /*小于 ICMP 报头长度则不合理*/
{
printf("ICMP packets\'s length is less than 8\n");
return -1;
}
/*确保所接收的是我所发的 ICMP 的回应*/
if( (icmp->icmp_type==ICMP_ECHOREPLY) && (icmp->icmp_id==pid) )
{
tvsend=(struct timeval *)icmp->icmp_data;
tv_sub(&tvrecv,tvsend); /*接收和发送的时间差*/
```

```
rtt=tvrecv.tv_sec*1000+tvrecv.tv_usec/1000; /*以毫秒为单位计算 rtt*/
/*显示相关信息*/
printf("%d byte from %s: icmp_seq=%u ttl=%d rtt=%.3f ms\n",len,
inet_ntoa(from.sin_addr), icmp->icmp_seq, ip->ip_ttl, rtt);
}
else
        return -1;
}

/*主函数*/
main(int argc,char *argv[])
{
struct hostent *host;
struct protoent *protocol;
unsigned long inaddr=0l;
int waittime=MAX_WAIT_TIME;
int size=50*1024;

if(argc<2)
{
printf("usage:%s hostname/IP address\n",argv[0]);
exit(1);
}
if( (protocol=getprotobyname("icmp") )==NULL)
{
perror("getprotobyname");
exit(1);
}
/*生成使用 ICMP 的原始套接字，这种套接字只有 root 才能生成*/
if( (sockfd=socket(AF_INET,SOCK_RAW,protocol->p_proto) )<0)
{
perror("socket error");
exit(1);
}
/* 回收 root 权限，设置当前用户权限*/
setuid(getuid());
/*扩大套接字接收缓冲区到 50KB，这样做主要为了减小接收缓冲区溢出的可能性，若无意中 ping 一个广
播地址或多播地址，将会引来大量应答*/
setsockopt(sockfd,SOL_SOCKET,SO_RCVBUF,&size,sizeof(size) );
bzero(&dest_addr,sizeof(dest_addr));
dest_addr.sin_family=AF_INET;
/*判断是主机名还是 IP 地址*/
if( inaddr=inet_addr(argv[1])==INADDR_NONE)
{
if((host=gethostbyname(argv[1]) )==NULL) /*是主机名*/
{
perror("gethostbyname error");
exit(1);
}
memcpy( (char *)&dest_addr.sin_addr,host->h_addr,host->h_length);
```

```
}
else    /*是 IP 地址*/
memcpy( (char *)&dest_addr,(char *)&inaddr,host->h_length);
/*获取 main 的进程 id，用于设置 ICMP 的标识符*/
pid=getpid();
printf("PING %s(%s): %d bytes data in ICMP packets.\n",argv[1],
inet_ntoa(dest_addr.sin_addr),datalen);
send_packet(); /*发送所有 ICMP 报文*/
recv_packet(); /*接收所有 ICMP 报文*/
statistics(SIGALRM); /*进行统计*/
return 0;
}

/*两个 timeval 结构相减*/
void tv_sub(struct timeval *out,struct timeval *in)
{
if( (out->tv_usec-=in->tv_usec)<0)
{
--out->tv_sec;
out->tv_usec+=1000000;
}
out->tv_sec-=in->tv_sec;
}
/*------------ The End -----------*/
```

使用 gcc 编译 myping.c，并生成可执行文件 myping：

```
#gcc –o myping myping.c
```

提 示

只有 root 用户才能利用 socket()函数创建原始套接口，所以在编译此程序的时候必须以 root 身份登录，以让可执行程序 myping 属于 root 用户。

运行程序，这里还是去 ping 局域网中的那台主机(192.168.1.103)，以使读者方便对比，如下所示为程序运行的输出结果：

```
#./myping 192.168.1.103
PING 192.168.1.103 (192.168.1.103) 56 bytes of data in ICMP packets.
64 bytes from 192.168.1.103: icmp_seq=1 ttl=64 time=3028.000ms
64 bytes from 192.168.1.103: icmp_seq=2 ttl=64 time=2018.000ms
64 bytes from 192.168.1.103: icmp_seq=3 ttl=64 time=1010.000ms

------------------PING statistics------------------
3 packets transmitted, 3 received, 0% loss
```

由于 myping.c 是发送完所有的 ICMP 报文才去接收，因此，第一、第二和第三个 ICMP 报文的往返时间依次大约是 3 秒，2 秒，1 秒，上述结果中 rtt 信息正反映了这一事实。

12.6 本章小结

网络编程是 Linux 下程序开发的重要内容，本章重点讲述了 Linux 网络编程的原理和方法。首先介绍了计算机网络体系结构的基本知识，这是网络编程的基础，读者务必理解计算机网络的协议模型。接着介绍了套接口的概念、套接口的数据结构及一些常用的操作套接口的函数。最后分别介绍了 3 种套接口(TCP 套接口、UDP 套接口和原始套接口)的编程原理，并以具体的实例详细说明了它们各自的使用。

实战演练

1. 编写一个程序，首先在内存中申请一段连续的存储空间，比如可以定义一个长度为 10 的数组，然后将这段连续内存单元中的值均设置为整数 2010。

2. 编写一个程序，使用 IP 地址转换函数 inet_aton()将本机的 IP 地址转换为网络字节序，并打印转换后的结果。本机 IP 地址由程序自动获得。

3. 编写一个程序，使用 gethostbyname()函数实现本地主机名到 IP 地址的转换，并打印转换的结果。

4. 编写一个程序，使用 socket()函数创建一个 TCP 套接口，并返回该套接字的描述符。

5. 编写一个程序，使用 bind()函数在一个打开的 socket 上面绑定 IP 地址与端口号，绑定的 IP 地址为本地主机 IP，绑定的端口为 5555。

6. 编写一个程序，使用 connect 函数将本地套接口连接到远程服务器，比如 www.sina.com 网站(IP 为 59.175.132.70)的服务器。

7. 编写一个程序，使用 socket()函数创建一个 UDP 套接口，并返回该套接字的描述符。

8. 编写程序，使用 sendto()函数和 recvfrom()函数在基于 UDP 套接口的客户机与服务器程序中实现简单数据的发送与接收。

9. 编写一个程序，使用 socket()函数创建一个原始套接口，并返回该套接字的描述符。

第13章

Linux图形界面编程

近些年来，人们使用计算机的方式已经发生很大的变化。最初是在一个黑屏幕上使用一些神秘的命令，这些命令种类繁多、数量庞大，普通用户使用很不方便，但经过近 10 年的发展，计算机已经进入了图形界面的环境，甚至于任何一个应用软件都可以做出很漂亮美观的用户界面。现在很少有人再考虑使用基于文本的程序。因此，对于今天的用户来说，图形用户界面已经是成功开发应用程序所必需的。

本章将介绍 Linux 下的图形界面编程，重点介绍基于 C 语言的具有面向对象特征的 GTK+图形界面编程。主要介绍 GTK+图形界面应用程序的框架、基本原理、常用元件的使用等。



本章内容：

◎ Linux 下的图形界面编程简介。
◎ GTK+界面编程的基本元件及其他常用元件。
◎ GTK+界面布局元件。
◎ GTK+的信号与回调函数。

13.1 Linux 下的图形界面编程简介

图形用户界面(Graphical User Interface，简称 GUI，又称图形用户接口)是指采用图形方式显示的计算机操作用户界面。与早期计算机使用的命令行界面相比，图形界面对于用户来说在视觉上更易于接受。GUI 的组成部分包括桌面、视窗、单一文件界面(Single Document Interface)、多文件界面(Multiple Document Interface)、标签、菜单、图表、按钮等。

GUI 的广泛应用是当今计算机发展的重大成就之一，它极大地方便了非专业用户的使用，人们从此不再需要死记硬背大量的命令，取而代之的是通过窗口、菜单、按键等方式来方便地进行操作。

程序员们用来进行图形用户界面编程的工具(或库)称为 GUI 工具包(或 GUI 库)，GUI 库是构造图形用户界面(程序)所使用的一套按钮、滚动条、菜单和其他对象的集合。在 UNIX 系统里，有很多可供使用的 GUI 库，如 Motif、Qt、GTK+、wxWindows、XForms 等，其中较为常用的是 Qt 和 GTK+。

本章将重点介绍 GTK+函数库。

13.1.1 Qt 简介

Qt 是一个跨平台的图形用户界面开发库，它不仅支持 Linux 操作系统，还支持所有类型的 UNIX 及 Windows 操作系统。

Qt 是一个 C++工具包，它由几百个 C++类构成，程序员在程序中可以使用这些类。因为 C++是面向对象的编程(Object-Oriented Programming，OOP)语言，而 Qt 是基于 C++构造的，所以，Qt 也具有 OOP 的所有优点。

Qt 良好的封装机制使它的模块化程度非常高，可重用性很强，Qt 提供了丰富的 API 供开发人员使用。使用 Qt 开发的图形用户界面程序具有良好的稳定性和健壮性。桌面环境 KDE(K Desktop Environment，即 K 桌面环境)就是使用 Qt 作为其底层库开发出来的。

13.1.2 GTK+简介

由于 Qt 使用 C++面向对象编程语言作为其开发语言，而许多在 Linux 下从事开发的程序员更喜欢或更习惯于用 C 语言。GTK+使用 C 语言作为开发语言，它基于 LGPL 授权，因此 GTK+是开放源代码而且完全免费的。Linux 的桌面环境 GNOME 就是建立在 GTK+的基础上。

简单地说，GTK+就是用 C 语言编写的用于开发图形界面程序的函数库。GTK+来源于 GIMP(GNU Image Manipulation Program，即 GNU 图像处理程序)。GTK+在 GDK(GIMP Drawing Kit，即 GIMP 绘图包)基础上创建，对它进行封装。GTK+简单易用，它设计良好，灵活而富有扩展性。它是自由软件，这意味着它不仅开放源代码，而且还可以免费使用。由于它使用 C 语言作为其开发语言，而 C 语言是跨平台的，因此 GTK+几乎可以在任何操作系统上使用。

图 13.1 显示了 GTK+在几种相关的开发库中的位置。图中每层除了与其上下相邻的两层有联系外，似乎与其他层没有关系。实际上，任何上层都可以调用位于它下面的各层提供的函数。例如，GTK+不仅可以调用 GDK 函数，也可以调用 glib 和 C 库函数。

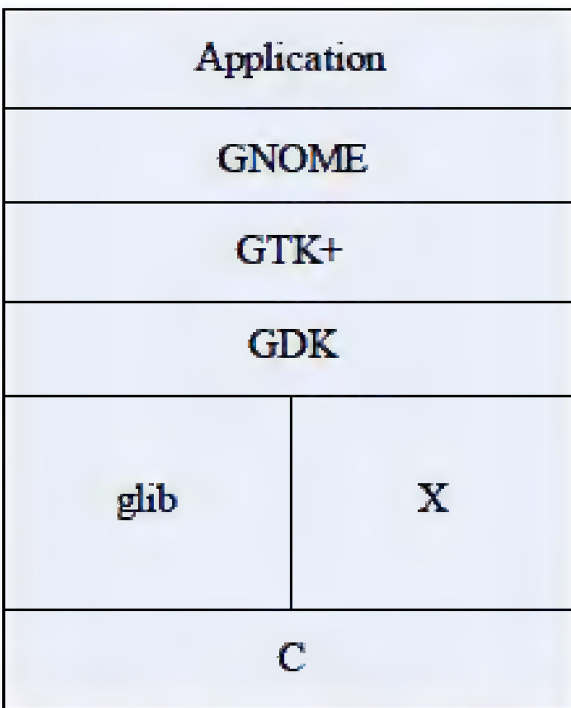


图 13.1　GTK+在几种相关的开发库中的位置

表 13.1 按层说明了上图各函数库的具体含义和功能。

表 13.1　各层函数库的具体描述

层	具 体 描 述
C	有两类 C 库函数可供调用，一类是标准 C 的库函数，如 printf、scanf；另一类是 Linux 的系统调用，如 open、read、write、fork
glib	glib 是 GDK、GTK+、GNOME 应用程序常用的库。它包含内存分配、字符串操作、日期和时间、定时器等库函数，也包括链表、队列、树等数据结构相关的工具函数
X	它是控制图形显示的底层函数库，包括所有的窗口显示函数、响应鼠标和键盘操作的函数
GDK	GDK(GIMP 绘图包)是为了简化程序员使用 X 函数库而开发的。X 库是其低层函数库，GDK 对其进行了包装，从而使程序员的开发效率大为提高
GTK+	GTK+就是 GIMP 工具包，它把 GDK 提供的函数组织成对象，使用 C 语言模拟出面向对象的特征，这使得用它开发出来的图形界面程序更为简单和高效。GTK+的一个重要组成部分是 widget(控件，也称为小部件)，按钮、文本编辑框、标签等都是 widget
GNOME	GNOME 库是对 GTK+的扩展，GNOME 桌面环境用来控制整个桌面。GNOME 使用 GNOME 对象和函数与桌面小部件交互，基本小部件由 GTK+处理。GNOME 为了方便程序员还增加了一些专门的小部件
Application	Application 即应用程序，它完成窗口的初始化，创建并显示窗口，进入消息循环，等待用户使用鼠标或键盘进行操作

最初，GTK+是作为另一个著名的开放源码项目 GIMP 的副产品而创建的。在开发早期的 GIMP 版本时，开发人员创建了 GTK(GIMP Toolkit)作为 Motif 工具包的替代，后者在那个时候不是免费的。当 GTK 这个工具包获得了面向对象特性和可扩展性之后，才在名称后面加上了一个加号，即现在的 GTK+。

十年过去了，GIMP 无疑仍然是使用 GTK+的最著名的程序之一，不过现在它已经不是唯一使用 GTK+库的程序了。GNU 的开发人员已经为 GTK+编写了成百上千的应用程序，而且至少有两个主要的桌面环境(Xfce 和 GNOME)用 GTK+为用户提供完整的工作环境。

GTK+具有以下优点：

(1) 简单易用

这一点应当很明显，但是它实际上含义丰富。工具包对于用户来说应当易用，这样才有可能创建简单的、直觉的和乐于使用的界面，哪怕针对的是新手。创建人机交互的正确模型不是

一项简单的任务，GTK+正是长时间工作的结果，而且是众多的甚至困难的决策的结果。

GTK+对于开发人员也易于使用。它允许开发人员用简单的方式说出自己想要的东西，不会用所谓的正规方式给开发人员带来负担，这些正规方式是计算机为了弥补它们固有的缺乏想象力的缺陷而施加给人类的负担。

(2) 设计良好、灵活和可扩展性

编写 GTK+的方式允许在不扭曲基本设计的情况下，让维护人员添加新功能、让用户利用新功能。工具包也是可扩展的，这意味着可以向其中添加自己的块，并用使用内置块一样的方式使用它们。例如，可以编写自己的控制元素，比如说用于显示应用程序处理的科学数据，并让它正确地遵照用户选择的显示风格，就像 GTK+自身的控件那样。

更进一步，GTK+是可定制的，这样就可以让它适应自己的需求。GTK+有一个系统，可以在所有应用程序之间复制设置，包括主题的选择。主题是一组一同发布的定制设置，会影响 GTK+使用的基本控件看起来的效果，甚至是某种程度上的行为方式。使用主题，可以模拟另一个操作系统的观感。

(3) 带有自由开放源码许可的自由软件

自由软件意味着每个人不仅可以自由地获得和使用这个工具包，还可以在满足某些条件的情况下修改并重新发布它。自由开放源码许可意味着这些条件不是严格限制的，可以得到的自由程度是显著的。最重要的是，GTK+采用了 Lesser General Public License(LGPL)许可，这是 GNU 许可家族中一个不太严格的许可。LGPL 允许自由地获取、修改和发布它覆盖的任何软件，只要对修改也保持自由即可。LGPL 还允许任何用户使用该库提供的功能，而不要求用户公开应用程序代码。这对于许多工业应用来说很重要，因为由于以前的协议或许可，这种场合下一般不希望公开代码或者公开代码是显然不现实的。

(4) 可移植性

最后，GTK+是可移植的。这意味着用户可以在许多平台和系统上运行它。另一方面，开发人员可以把软件提供给众多用户，只要编写一次程序，还可以使用许多不同的编程和开发平台、工具和编程语言。所有这些都可以理解为更多的潜在用户，程序开发者可以利用这一点更好地满足需求更广泛的技能和工具。

所有这些优势组合在一起，让 GTK+成为图形界面软件开发的坚实基础。有了它，就能够把注意力集中在解决实际问题上，而不必重新描述每一个底层函数。

在安装 Fedora Core 或者 Red Hat Linux 系列操作系统时，如果选择了安装应用程序开发包，那么操作系统安装完毕后，GTK+开发包就已经安装好了。如果没有安装，请从网络上(http://www.gtk.org)免费下载一份 GTK 源代码并安装到系统上，也可以插入 Linux 安装光盘在系统提示下进行安装。由于安装过程非常简单，这里就不再赘述了。

13.2 界面基本元件

本节通过一个简单常见的例子向读者介绍 GTK+图形界面编程中的基本元件(有些书籍中称为控件，为便于读者理解，本书统一称为元件)，以及创建和操作这些元件的函数调用。基本元

件包括窗口、标签、按钮、文本框等。

13.2.1 一个简单的例子

在介绍 GTK+图形界面编程的基本元件之前，我们先来看一个关于这些基本元件使用的既简单又常见的例子。如图 13.2 所示的界面，读者一定不会陌生，它是一个很常见的用户登录界面，在很多系统的开始都会使用这样的登录界面，用户在这里输入用户名和密码(当然，可能还有一些其他的信息)，系统对用户填入的信息进行身份验证。

在这个简单的界面中，一共包含了 4 种 GTK+基本元件：窗口、标签、按钮和文本框。当然，还使用了一个 GTK+界面布局元件 —— 表格(关于表格，将在下一节讲到)。整个界面为一个窗口元件，“用户名”和“密码”是标签元件，“取消”和“确定”是按钮元件，而用户填写用户名和密码的区域便是文本框元件。

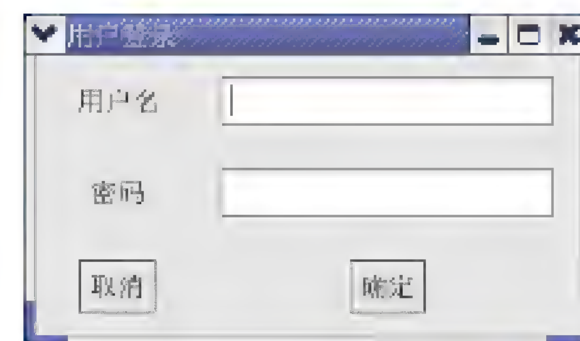


图 13.2 一个简单的界面

下面我们看看这个简单的界面用 GTK+程序是如何实现的。程序 13.1 给出了生成该界面的源代码，如 login.c 所示。为便于读者理解，程序中给出了尽可能详尽的注释，而关于各个函数的具体使用方法，将会在本章后续的内容中向读者一一介绍。

【程序 13.1】一个简单的用户登录界面：login.c。

```
#include <gtk/gtk.h>
int main (int argc,char *argv[])
{
    GtkWidget *window; /*指向窗口的指针*/
    GtkWidget *table; /*指向表格的指针*/
    GtkWidget *label1; /*指向标签 1 的指针*/
    GtkWidget *label2; /*指向标签 2 的指针*/
    GtkWidget *entry1; /*指向文本框 1 的指针*/
    GtkWidget *entry2; /*指向文本框 2 的指针*/
    GtkWidget *button1; /*指向按钮 1 的指针*/
    GtkWidget *button2; /*指向按钮 2 的指针*/
    gtk_init(&argc,&argv); /*初始化图形显示环境*/
    window=gtk_window_new (GTK_WINDOW_TOPLEVEL); /*新建窗口 window */
    gtk_window_set_title (GTK_WINDOW (window),
g_locale_to_utf8("用户登录",-1,NULL,NULL,NULL));
/*为窗口设置标题，g_locale_to_utf8()函数支持中文字符显示*/
    table=gtk_table_new (3,2,FALSE);
            /*定义一个 3 行 2 列的表格，单元格大小会根据单元格中的元件大小自动调整*/
label1=gtk_label_new (g_locale_to_utf8("用户名",-1,NULL,NULL,NULL));
/*新建“用户名”标签*/
    gtk_widget_set_size_request (label1, 38, 15); /*设置标签 1 的大小*/
    label2=gtk_label_new (g_locale_to_utf8("密码",-1,NULL,NULL,NULL));
/*新建“密码”标签*/
    gtk_widget_set_size_request (label2, 30, 15); /*设置标签 2 的大小*/
    entry1=gtk_entry_new ();/*新建用户名文本框*/
    entry2=gtk_entry_new ();/*新建密码文本框*/
    button1=gtk_button_new_with_label(g_locale_to_utf8("取消",-1,NULL,NULL,NULL));
/*新建“取消”按钮*/
    button2=gtk_button_new_with_label(g_locale_to_utf8("确定",-1,NULL,NULL,NULL));
```

```
/*新建"确定"按钮*/
    gtk_container_add (GTK_CONTAINER (window), table);
                                        /*将表格添加到窗口中*/
    /*下面是将 6 个元件分别添加到表格相应的位置中*/
    gtk_table_attach (GTK_TABLE(table), label1, 0, 1, 0, 1,
                (GtkAttachOptions)(0), (GtkAttachOptions)(0), 20, 10);
    gtk_table_attach (GTK_TABLE(table), entry1, 1, 2, 0, 1,
                (GtkAttachOptions)(0), (GtkAttachOptions)(0), 10, 10);
    gtk_table_attach (GTK_TABLE(table), label2, 0, 1, 1, 2,
                (GtkAttachOptions)(0), (GtkAttachOptions)(0), 20, 10);
    gtk_table_attach (GTK_TABLE(table), entry2, 1, 2, 1, 2,
                (GtkAttachOptions)(0), (GtkAttachOptions)(0), 10, 10);
    gtk_table_attach (GTK_TABLE(table), button1, 0, 1, 2, 3,
                (GtkAttachOptions)(0), (GtkAttachOptions)(0), 20, 10);
    gtk_table_attach (GTK_TABLE(table), button2, 1, 2, 2, 3,
                (GtkAttachOptions)(0), (GtkAttachOptions)(0), 10, 10);
    gtk_widget_show (window);     /*依次显示窗口中的所有元件*/
    gtk_widget_show (table);
    gtk_widget_show (label1);
    gtk_widget_show (label2);
    gtk_widget_show (entry1);
    gtk_widget_show (entry2);
    gtk_widget_show (button1);
    gtk_widget_show (button2);
    gtk_main();
    return 0;
}
```

使用 gcc 编译 login.c，生成可执行文件 login，命令形式如下：

```
#gcc –o login login.c `pkg-config --libs --cflags gtk+-2.0`
```

运行程序，得到输出结果：

```
# ./login
```

提 示

编译命令中的字符串"pkg-config--libs--cflags gtk+-2.0"两边是反引号(在键盘上位于数字字符 1 的左边)。

下面向读者介绍这些基本元件的创建与使用方法。

13.2.2 窗口

窗口是一个应用程序的界面框架，程序的所有内容和与用户的交互都在这个窗口中。在设置应用程序的界面时，第一步是建立一个窗口。

1. 新建一个窗口

gtk_window_new()函数可以建立一个 GTK+窗口，函数使用方法如下：

```
#include <gtk/gtk.h>
GtkWidget * gtk_window_new (GtkWindowType type);
```

返回：若成功，则返回一个 GtkWidget 类型的指针；若失败，则返回空指针 NULL。

参数 type 是一个表示窗口状态的常量，可能的取值有以下两种：

- GTK_WINDOW_TOPLEVEL：表示这个窗口是一个正常的窗口。窗口可以最小化，最小化以后，在窗口管理器中可以看到这一窗口的按钮。窗口管理器相当于 Windows 下的任务栏。
- GTK_WINDOW_POPUP：表示这个窗口是一个弹出式窗口，不可以最小化。但这个窗口是一个独立运行的程序，并不是一个对话框。

gtk_window_new()函数的返回值是一个 GtkWidget 类型的指针。读者在后面将会看到，图形界面的所有元素都会返回给这个指针。GtkWidget 结构体的定义方式如下：

```
typedef struct
{
    GtkStyle *style;                /*元件的风格*/
    GtkRequisition requisition; /*元件的位置*/
    GtkAllocation allocation;   /*元件允许使用的空间*/
    GtkWindow *window;         /*元件所在的窗口或父窗口*/
    GtkWidget *parent;           /*元件的父窗口*/
} GtkWidget;
```

新建一个窗口以后，这个窗口还不会马上显示出来，需要调用窗口显示函数 gtk_widget_show()来显示这个窗口，函数原型如下：

```
#include <gtk/gtk.h>
void gtk_widget_show (GtkWidget * widget);
```

参数 widget 是一个 GtkWidget 类型的结构体指针，指向需要显示的窗口。gtk_widget_show 函数无返回值。

2. 设置窗口标题

gtk_window_set_title()函数用于设置窗口的标题，函数原型如下：

```
#include <gtk/gtk.h>
void gtk_window_set_title (GtkWindow *window, gchar *title);
```

参数 window 指定将要添加标题的窗口，title 表示需要设置的标题。函数无返回值。

需要提醒读者的是，指针 title 所指向的字符串必须是英文字符的，这是由于 C 语言的基本规则所限定的，但如果我们想要显示中文字符，则必须像程序 13.1 那样，使用 g_locale_to_utf8()函数进行批注。

下面这段代码创建了一个最原始的窗口(窗口中无任何其他元件)，并为窗口设置了一个英文标题。可以看到，它使用了很多在程序 13.1 中见过的函数。程序源代码如 first_win.c 所示。

【程序 13.2】创建一个最原始的 GTK+窗口，并为窗口设置标题：first_win.c。

```
#include <gtk/gtk.h>
int main (int argc,char *argv[])
```

```
{
GtkWidget *window;              /*定义一个指向窗口的指针*/
char title[]="My first Window";     /*窗口的标题*/
gtk_init(&argc,&argv);
/*初始化图形显示环境，主函数的参数可以带入这个函数中，作为新建窗口的参数*/
window = gtk_window_new (GTK_WINDOW_TOPLEVEL); /*新建窗口 window */
gtk_window_set_title (GTK_WINDOW (window), title);        /*为窗口设置标题*/
gtk_widget_show (window);    /*显示窗口*/
gtk_main();                     /*进入消息处理循环*/
return 0;
}
```

另外，在介绍 GTK+的其他函数之前，为便于读者理解和后续内容的介绍，这里先将 GTK+预定义的函数和数据类型说明如下。GTK+预定义的函数前缀说明如表 13.2 所示。

表 13.2　GTK+预定义函数前缀的含义

前　缀	含　义
G	glib 定义的数据结构
g	glib 声明的数据类型
g_	glib 定义的函数
gtk_	GTK+定义的函数
Gtk	GTK+库的对象或数据结构
GTK	GTK+定义的宏或者常量

GTK+预定义数据类型说明如表 13.3 所示。

表 13.3　GTK+预定义的数据类型说明

GTK+的数据类型	C 语言数据类型
gchar	char
gint	int
glong	long
gboolean	char
gfloat	float
gdouble	double
guchar	unsigned char
guint	unsigned int
gulong	unsigned long
gpointer	void *
gint8	在任何平台上都是 8 位的整型
gint16	在任何平台上都是 16 位的整型

(续表)

GTK+的数据类型	C 语言数据类型
gint32	在任何平台上都是 32 位的整型
guint8	在任何平台上都是 8 位的无符号整型
guint16	在任何平台上都是 16 位的无符号整型
guint32	在任何平台上都是 32 位的无符号整型

3. 设置窗口大小与位置

窗口的大小指的是窗口的宽度和高度，可以用 gtk_widget_set_usize()函数来设置一个窗口的初始大小。窗口的位置指的是窗口的左上顶点到屏幕左边和顶边的距离，可以用 gtk_widget_set_uposition()函数来设置一个窗口的初始位置。这两个函数的原型如下：

```
#include <gtk/gtk.h>
void gtk_widget_set_usize (GtkWidget * widget, gint width, gint height);
void gtk_widget_set_uposition (GtkWidget * widget, gint x, gint y);
```

参数 widget 用于指定将要进行设置的窗口；width 表示窗口的宽度；height 表示窗口的高度；x 表示窗口的左边距，也就是窗口左上顶点的 x 坐标，y 表示窗口的上边距，也就是窗口左上顶点的 y 坐标。两个函数均无返回值。

例如设置新建窗口的宽度为 400，高度为 200，窗口的左边距为 200，上边距也为 200(注意它们的单位均是“像素点”)，可以在程序中加上下面的代码段：

```
gtk_widget_set_usize (window, 400, 200);        /*设置窗口的大小*/
gtk_widget_set_uposition (window, 200, 200);    /*设置窗口的位置*/
```

生成窗口以后，窗口的初始大小与位置即为程序中设定的参数，拖动鼠标仍然可以改变它的大小或位置，但发现窗口在改变大小时，不会小于它的初始大小。

13.2.3 标签

标签是程序中的一个文本，这个文本可以显示一定的信息，但是用户不能改变和输入标签的内容。通常，界面中的提示信息或显示内容都是通过标签来实现的，例如图 13.2 中的“用户名”和“密码”两个标签。

1. 新建一个标签

在窗口中使用标签之前，需要新建一个标签。gtk_ label_new()函数用于新建一个标签，函数原型如下：

```
#include <gtk/gtk.h>
GtkWidget *gtk_ label_new (gchar *text);
```

返回：若成功，则返回一个 GtkWidget 类型的指针；若失败，则返回空指针 NULL。

参数 text 表示标签中需要显示的内容。如同窗口一样，新建标签后，需要调用标签显示函数 gtk_widget_show()来显示这个标签，函数原型如下：

```
#include <gtk/gtk.h>
void gtk_widget_show (GtkWidget *label);
```

参数 label 表示 gtk_ label_new()函数新建的标签，gtk_widget_show()函数无返回值。

2. 将标签添加到窗口

在 GTK+窗口中，除了 Window 窗口外，其他的任何元件都必须放置在一个容器中。例如新建的标签并不能直接显示，而需要放在一个窗口元件中。gtk_container_add()函数的作用是把一个元件放置在另一个元件窗口(容器)中，函数原型如下：

```
#include <gtk/gtk.h>
void gtk_container_add (GtkContainer *container, GtkWidget *widget);
```

参数 container 是一个父级容器指针，widget 是需要放置的元件的指针。该函数无返回值。

在程序 13.1 中，我们使用了函数 gtk_table_attach()将标签添加到表格(容器)中，因为表格对于标签来说，便是一个父级容器。然后再使用 gtk_container_add()函数将表格添加到窗口(容器)中，因为窗口对于表格来说，又是表格的一个父级容器。

3. 设置和获取标签的文本

标签上的文本内容是程序员在开发时进行设置的，界面开发完成后，普通用户是无法改变标签的文本内容的。在程序中，可以使用 gtk_label_get_text()函数来获取一个标签的文本，使用 gtk_label_set_text()函数来设置一个标签的文本。它们的函数原型如下：

```
#include <gtk/gtk.h>
const char *gtk_label_get_text (GtkLabel *label);
void gtk_label_set_text (GtkLabel *label, gchar *text);
```

第一个函数返回：若成功，则返回标签文本的字符串指针；若失败，则返回空指针 NULL。第二个函数无返回值。

函数中的参数 label 是一个指向标签的指针，text 表示标签需要设置的文本。

13.2.4 按钮

在图形界面的程序中，有很多操作都是通过窗口程序的按钮来实现的。在后面我们还将看到按钮最常用于发送的一个信号，这个信号会引起相应事件的响应。

1. 新建一个按钮

函数 gtk_button_new_with_label()用来新建一个带有标签的按钮，函数原型如下：

```
#include <gtk/gtk.h>
GtkWidget *gtk_button_new_with_label (gchar *label);
```

返回：若成功，则返回一个 GtkWidget 类型的指针；若失败，则返回空指针 NULL。

参数 label 表示一个字符串，这个字符串会显示在按钮上，称为按钮的标签。同样地，新建按钮成功后，需要调用 gtk_widget_show()来显示这个按钮。

2. 设置和获取按钮的标签

按钮的标签指的是按钮上的文字。函数 gtk_button_get_label()可以获取某个按钮的标签，函数 gtk_button_set_label()可以设置某个按钮的标签，这两个函数的原型如下：

```
#include <gtk/gtk.h>
const gchar *gtk_button_get_label (GtkButton *button);
void gtk_button_set_label (GtkButton *button, const gchar *label);
```

第一个函数返回：若成功，则返回按钮的标签内容的字符串指针；若失败，则返回空指针 NULL。第二个函数无返回值。

函数中的参数 button 是一个指向按钮的指针，label 表示按钮的标签内容。函数 gtk_button_get_label()会取得这个按钮的标签作为一个字符串返回；gtk_button_set_label()则会把参数 label 指向的字符串设置成按钮的标签。

按钮的使用通常伴随着 GTK+信号与事件的产生，我们将在 13.5 节中看到一个这样的例子。

13.2.5 文本框

文本框是界面中的输入区域，用户可以在这个区域中用键盘输入内容，界面程序中的各种输入都是通过文本框来完成的，例如图 13.2 中的用户名和密码的输入文本框。

1. 新建文本框

函数 gtk_entry_new()用来新建一个文本框，函数原型如下：

```
#include <gtk/gtk.h>
GtkWidget *gtk_entry_new (void);
```

返回：若成功，则返回一个 GtkWidget 类型的指针；若失败，则返回空指针 NULL。

另一个建立文本框的函数为：

```
GtkWidget *gtk_entry_new_with_max_length (gint max);
```

返回：同上。

在这个函数中，有一个参数 max，用来表示这个文本框最多可以输入的字符数。如果已经输入这些数目的字符，就不能再向文本框中输入内容了。

2. 设置和获取文本框数据

在文本框中输入数据以后，经常需要取得这些数据进行相关的处理；新建文本框时可以设置文本框的初始内容。函数 gtk_entry_get_text()和 gtk_entry_set_text()可以分别完成这两个功能。它们的函数原型如下：

```
#include <gtk/gtk.h>
const gchar *gtk_entry_get_text (GtkEntry *entry);
void gtk_entry_set_text (GtkEntry *entry, const gchar *text);
```

第一个函数返回：若成功，则返回指向文本框中的字符串的指针；若失败，则返回空指针 NULL。第二个函数无返回值。

在参数列表中，entry 是一个指向文本框的指针，text 表示需要设置到文本框中的字符串文本。

gtk_entry_get_text()是在 GTK+图形界面中经常使用到的一个函数，它通常用于获取用户在文本框中输入的字符串信息，例如在 13.2 节最开始的那个例子中，我们可以使用 gtk_entry_get_text() 函数获取“用户名”和“密码”文本框中输入的信息，以获取用户的用户名和密码。

程序 13.3 是关于使用 GTK+文本框和按钮的例子。程序中先新建了一个具有输入字数限制的文本框，并将这个文本框添加到表格中，当用户单击“提交”按钮时调用 gtk_entry_get_text() 函数获取文本框中的字符串信息，然后在终端打印输出这些字符串。程序源代码如 entry_example.c 所示。

【程序 13.3】GTK+文本框的使用：entry_example.c。

```
#include <gtk/gtk.h>
#include <stdlib.h>

GtkWidget *window;
GtkWidget *table;
GtkWidget *entry;   /*定义一个指向文本框的指针*/
GtkWidget *label;
GtkWidget *button;
char text[50];

void on_clicked(GtkWidget *widget, gpointer data)   /*定义回调函数*/
{
    strcpy(text, gtk_entry_get_text (GTK_ENTRY (entry)));
      /*获取文本框的文本内容，并将其复制到 text 字符串数组中*/
    printf("您输入的字符串是: %s\n",text);   /*打印文本框中输入的字符串*/
}

int main (int argc,char *argv[])
{
    gtk_init(&argc,&argv);
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window),
g_locale_to_utf8("文本框的使用",-1,NULL,NULL,NULL));
                        /*为窗口设置标题，g_locale_to_utf8()函数支持中文字符显示*/
    table=gtk_table_new (3,2,FALSE);
               /*定义一个 3 行 2 列的表格，单元格大小会根据单元格中的元件大小自动调整*/
    label=gtk_label_new (g_locale_to_utf8("请在这里输入文本:",-1,NULL,NULL,NULL));
    entry = gtk_entry_new_with_max_length (50);
              /*新建一个有字数限制的文本框，文本框中最多可输入 50 个字符*/
    button=gtk_button_new_with_label (g_locale_to_utf8("提交",-1,NULL,NULL,NULL));
/*新建“提交”按钮*/
    gtk_container_add (GTK_CONTAINER (window), table); /*将表格添加到窗口中*/
    /*下面是将 3 个元件分别添加到表格相应的位置中*/
    gtk_table_attach (GTK_TABLE(table), label, 0, 1, 0, 1,
                        (GtkAttachOptions)(0), (GtkAttachOptions)(0), 10, 10);
    gtk_table_attach (GTK_TABLE(table), entry, 0, 2, 1, 2,
```

```
                        (GtkAttachOptions)(0), (GtkAttachOptions)(0), 10, 10);
    gtk_table_attach (GTK_TABLE(table), button, 1, 2, 2, 3,
                        (GtkAttachOptions)(0), (GtkAttachOptions)(0), 10, 10);
    g_signal_connect (G_OBJECT(button), "clicked", G_CALLBACK(on_clicked), window);
                                    /*为“提交”按钮添加信号回调函数*/
    gtk_widget_show_all (window);   /*显示窗口中的所有元件*/
    gtk_main();
    return 0;
}
```

使用 gcc 编译 entry_example.c，生成可执行文件 entry_example，命令形式如下：

```
#gcc –o entry_example entry_example.c `pkg-config --libs --cflags gtk+-2.0`
```

运行程序，得到的输出结果如图 13.3 所示。

```
#./ entry_example
```

在文本框输入“I like Linux C programs!”字符串信息，然后单击“提交”按钮，如图 13.4 所示。

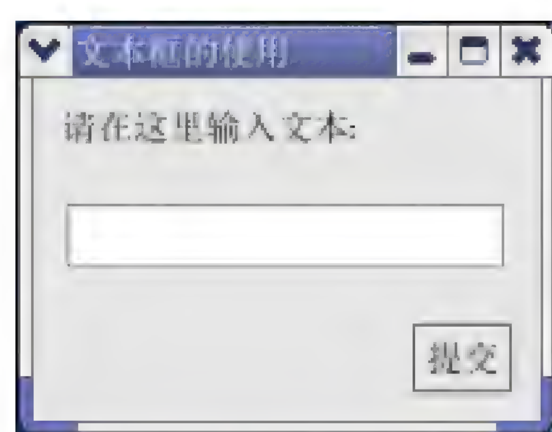


图 13.3　GTK+文本框的使用

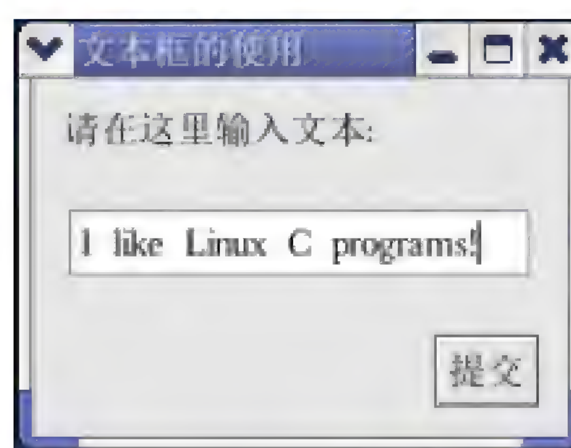


图 13.4　在文本框输入信息

单击“提交”按钮后，可以看到在终端 shell 中打印出了下列信息：

```
您输入的字符串是: I like Linux C programs!
```

这说明在信号回调函数中，已成功地使用 gtk_entry_get_text()函数获取到了文本框中的字符串信息。

在程序 13.3 中，读者看到了一个陌生的函数 g_signal_connect()，事实上，在单击“提交”按钮时，产生了相应的信号，系统自动调用相应的信号处理函数，而在本程序中，信号处理函数的作用正是在终端打印输出文本框中的字符串信息。13.5 节将向读者介绍 GTK+中的信号与信号处理函数。

另外，对于该文本框中的数据，用户是可以进行随意输入和删除操作的，但输入数据时长度不能超过程序 13.3 中的字数限制，即 50 个字符(包含空格)。

提 示

在程序 13.3 的最后使用了 gtk_widget_show_all(window)函数来显示窗口中的所有元件，这比程序 13.1 中介绍的依次显示窗口元件方便多了。

13.3 界面布局元件

本节将向读者介绍 GTK+图形界面编程中的界面布局元件，包括表格、框、窗格等，其中表格是在界面编程中最常用的布局元件，通过在表格的单元格中插入不同的元件，来实现元件的布局和排列。使用界面布局元件，可以在一个窗口中设计出复杂而优美的界面。

13.3.1 表格

表格是指用横竖布局的线和格子将一个窗口划分成多个区域，每个区域可以放置不同的元件。如果一个元件中可以存放其他的元件，这个元件就被称作容器。GTK+的容器都是二进制的，也就是说每个容器只能放置一个元件，如果想在一个窗口中放置多个元件，则需要使用表格、窗格等有多个单元格的容器。

1. 表格的建立

若想在窗口中使用表格，需要调用 gtk_table_new()函数新建一个表格，函数原型如下：

```
#include <gtk/gtk.h>
GtkWidget *gtk_table_new (guint rows, guint columns, gboolean homogeneous);
```

返回：若成功，则返回一个 GtkWidget 类型的指针；若失败，则返回空指针 NULL。

函数中，参数 rows 表示表格的行数，columns 表示表格的列数，它们的数据类型都是无符号整型(参考表 13.3)。需要注意的是，这里的行和列的编号是从边开始算起的，编号方法如图 13.5 所示。

图 13.5　表格行和列的编号方法

例如，某个元件占据了如图 13.5 中的阴影部分，则它所占格子的坐标为 left(左)为 0，right(右)为 2，top(上)为 1，bottom(下)为 3。

参数 homogeneous 是一个布尔值，如果设置为 TRUE，则每一个单元格的大小相同，所有单元格的高度与宽度和表格中最大的一个元件的高度与宽度相同；如果设置为 FALSE，则表格的单元格大小会根据单元格中的元件大小自动调整。

注意，表格的作用只是将窗口划分成不同的区域，并不能显示出这个表格，也就是说，在 GTK+图形界面编程中，表格是作为一个容纳其他元件的容器使用的，并不会实际显示。通过使用表格，可以把窗口中的其他元件放置到表格指定的行和列中。函数 gtk_table_attach()的作用是将一个元件添加到表格中，并且设置在表格中的位置和填充的选项。函数原型如下：

```
#include <gtk/gtk.h>
void gtk_table_attach (GtkTable *table, GtkWidget *child, guint left_attach,
guint right_attach, guint top_attach, guint bottom_attach,
GtkAttachOptions xoptions, GtkAttachOptions yoptions,
guint xpadding, guint ypadding);
```

返回：无。

函数中各个参数的含义和作用如下所示：

- table：容器表格的指针。
- child：需要添加的元件的指针。
- left_attach，right_attach：分别表示元件的左边是表格的第几条边，右边是表格的第几条边。需要注意的是，这里的边数是从 0 开始计算的。
- top_attach，bottom_attach：分别表示元件的上边是表格的第几条边，下边是表格的第几条边。
- xoptions，yoptions：分别表示元件在表格中的水平方向、垂直方向的对齐方式，取值类似为 GtkAttachOptions。
- xpadding，ypadding：分别表示元件与边框水平方向、垂直方向的边距。

另外，GtkAttachOptions 是 GTK+中用来描述元件对齐方式的变量，它的可能取值有下面 3 种情况：

- GTK_EXPAND：元件以实际设置的大小显示，如果大于容器的大小则容器自动变大。
- GTK_SHRINK：如果元件大于容器的大小，则自动缩小元件。
- GTK_FILL：元件填充整个单元格。

关于表格的使用，我们已在前面的程序示例中详细演示了，这里不再举例。

2. 合并单元格

合并单元格指的是一个元件占据一个表格中同行或同列的多个单元格。表格的合并没有专门的函数来实现，而是通过元件设置在表格中的位置时，设置它在表格中跨多个单元格的边界来实现的。

例如在程序 13.3 中，我们使用了这样的方法：

```
gtk_table_attach (GTK_TABLE(table), entry, 0, 2, 1, 2,
                  (GtkAttachOptions)(0), (GtkAttachOptions)(0), 10, 10);
```

让文本框元件占据了表格中第二行的两个单元格。

3. 嵌套表格

在设计较复杂的界面时，使用一个表格并不能完成布局界面中所有的元件，这时通常需要在表格的单元格中添加表格，即实现表格的嵌套。表格也是一个普通元件，可以把一个表格添加到另一个表格的单元格中，这样通过表格的嵌套，就可以实现复杂的布局方式。

程序 13.4 演示了合并单元格和嵌套表格的使用方法，程序设计了一个在填写用户信息时常见的界面。程序的功能是，首先创建一个 4 行 4 列的主表格，设置单元格大小会根据单元格中的元件大小自动调整，然后将各个元件放入表格中不同的位置。其中，第 1 行的第 1、2、3

列单元格合并为一个单元格，放置“我的信息”标签。第 1 行的第 4 列中放置一个嵌套的表格，嵌套表格为 1 行 2 列，分别放置了两个按钮，即“注册”和“登录”。另外，表格的第 3 行和第 4 行也分别进行了单元格的合并，即合并第 2、3、4 列的单元格，生成一个较大的文本框。程序源代码如 con_nested_table.c 所示。

【程序 13.4】创建嵌套表格，并合并单元格：con_nested_table.c。

```
#include <gtk/gtk.h>
int main (int argc,char *argv[])
{
    GtkWidget *window; /*指向窗口的指针*/
    GtkWidget *table1; /*指向表格 1 的指针*/
    GtkWidget *table2; /*指向表格 2 的指针*/
    GtkWidget *entry1; /*指向文本框 1 的指针*/
    GtkWidget *entry2; /*指向文本框 2 的指针*/
    GtkWidget *entry3; /*指向文本框 3 的指针*/
    GtkWidget *entry4; /*指向文本框 4 的指针*/
    GtkWidget *button1; /*指向按钮 1 的指针*/
    GtkWidget *button2; /*指向按钮 2 的指针*/
    GtkWidget *label1; /*指向标签 1 的指针*/
    GtkWidget *label2; /*指向标签 2 的指针*/
    GtkWidget *label3; /*指向标签 3 的指针*/
    GtkWidget *label4; /*指向标签 4 的指针*/
    GtkWidget *label5; /*指向标签 5 的指针*/

    gtk_init(&argc,&argv);
    window=gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window),
    g_locale_to_utf8("用户信息",-1,NULL,NULL,NULL));
                        /*为窗口设置标题，g_locale_to_utf8()函数支持中文字符显示*/
    table1=gtk_table_new(4,4,FALSE);
    /*定义一个 3 行 2 列的表格，单元格大小会根据单元格中的元件大小自动调整*/
    table2=gtk_table_new(1,2,FALSE);
    /*定义一个 1 行 2 列的表格 2，单元格大小会根据单元格中的元件大小自动调整*/
    /*下面是分别定义 5 个标签、2 个按钮、4 个文本框*/
    label1=gtk_label_new (g_locale_to_utf8("我的信息",-1,NULL,NULL,NULL));
    label2=gtk_label_new (g_locale_to_utf8("用户名",-1,NULL,NULL,NULL));
    label3=gtk_label_new (g_locale_to_utf8("密码",-1,NULL,NULL,NULL));
    label4=gtk_label_new (g_locale_to_utf8("电子邮件",-1,NULL,NULL,NULL));
    label5=gtk_label_new (g_locale_to_utf8("详细地址",-1,NULL,NULL,NULL));
    button1=gtk_button_new_with_label(g_locale_to_utf8("注册",-1,NULL,NULL,NULL));
    button2=gtk_button_new_with_label(g_locale_to_utf8("登录",-1,NULL,NULL,NULL));
    entry1=gtk_entry_new ();
    entry2=gtk_entry_new ();
    entry3=gtk_entry_new ();
    entry4=gtk_entry_new ();
    gtk_container_add (GTK_CONTAINER (window), table1); /*将表格添加到窗口中*/
    /*下面是将 5 个元件分别添加到表格相应的位置中*/
    gtk_table_attach (GTK_TABLE(table1), label1, 0, 3, 0, 1,
                        (GtkAttachOptions)(0), (GtkAttachOptions)(0), 5, 5);
```

```
    gtk_table_attach (GTK_TABLE(table1), table2, 3, 4, 0, 1,
                            (GtkAttachOptions)(0), (GtkAttachOptions)(0), 0, 5);
    gtk_table_attach (GTK_TABLE(table1), label2, 0, 1, 1, 2,
                            (GtkAttachOptions)(0), (GtkAttachOptions)(0), 5, 5);
    gtk_table_attach (GTK_TABLE(table1), entry1, 1, 2, 1, 2,
                            (GtkAttachOptions)(0), (GtkAttachOptions)(0), 5, 5);
    gtk_table_attach (GTK_TABLE(table1), label3, 2, 3, 1, 2,
                            (GtkAttachOptions)(0), (GtkAttachOptions)(0), 5, 5);
    gtk_table_attach (GTK_TABLE(table1), entry2, 3, 4, 1, 2,
                            (GtkAttachOptions)(0), (GtkAttachOptions)(0), 5, 5);
    gtk_table_attach (GTK_TABLE(table1), label4, 0, 1, 2, 3,
                            (GtkAttachOptions)(0), (GtkAttachOptions)(0), 5, 5);
    gtk_table_attach (GTK_TABLE(table1), entry3, 1, 4, 2, 3,
                            (GtkAttachOptions)(GTK_FILL), (GtkAttachOptions)(0), 5, 5);
    gtk_table_attach (GTK_TABLE(table1), label5, 0, 1, 3, 4,
                            (GtkAttachOptions)(0), (GtkAttachOptions)(0), 5, 5);
    gtk_table_attach (GTK_TABLE(table1), entry4, 1, 4, 3, 4,
                            (GtkAttachOptions)(GTK_FILL), (GtkAttachOptions)(0), 5, 5);
    /*下面是将两个元件分别添加到表格 2 相应的位置中*/
    gtk_table_attach (GTK_TABLE(table2), button1, 0, 1, 0, 1,
                            (GtkAttachOptions)(0), (GtkAttachOptions)(0), 2, 5);
    gtk_table_attach (GTK_TABLE(table2), button2, 1, 2, 0, 1,
                            (GtkAttachOptions)(0), (GtkAttachOptions)(0), 2, 5);
    gtk_widget_show_all (window); /*显示窗口中的所有元件*/
    gtk_main();
    return 0;
}
```

使用 gcc 编译 con_nested_table.c，生成可执行文件 con_nested_table，命令如下：

```
#gcc –o con_nested_table con_nested_table.c `pkg-config --libs --cflags gtk+-2.0`
```

运行程序，得到的输出结果如图 13.6 所示。

```
#./ con_nested_table
```

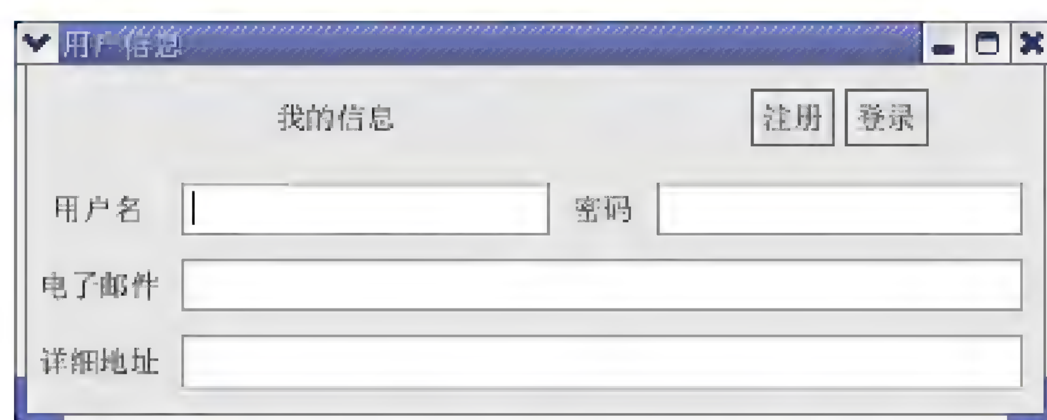


图 13.6　表格的合并与嵌套

13.3.2　框

在 GTK+中，框(box)是一种不可见的 widget 容器，它有水平框和垂直框两种。水平框是指元件按放入窗口的顺序水平排列，垂直框是指元件按放入窗口的顺序垂直排列。水平框可以看作是只有一行的表格，而垂直框可以看作是只有一列的表格，但是它们的操作比表格简单，放置元件时不需要考虑元件的位置。

1. 框的建立

水平框使用函数 gtk_hbox_new()生成，而垂直框使用函数 gtk_vbox_new()生成。它们的函数原型如下：

```
#include <gtk/gtk.h>
GtkWidget *gtk_hbox_new (gboolean homogeneous, gint spacing);
GtkWidget *gtk_vbox_new (gboolean homogeneous, gint spacing);
```

两个函数的返回：若成功，则返回一个 GtkWidget 类型的指针；若失败，则返回空指针 NULL。

参数 homogeneous 是一个布尔值，控制每个放入框中的元件是否有同样的高或宽，spacing 表示每一行元件之间的距离。

同样地，建立一个框后，需要调用 gtk_container_add()函数将这个框添加到窗口中，并且需要调用显示函数 gtk_widget_show()来显示这个框。

2. 在框中添加元件

同表格一样，框也是一个容器，当没有向这个容器中添加任何元件时，容器是不能显示的。函数 gtk_box_pack_start()或 gtk_box_pack_end()用于将元件放入框容器中。前者从左到右、从上到下将元件放入框容器，而后者相反，从右到左，从下到上将元件放入框容器中。它们的函数的原型如下：

```
#include <gtk/gtk.h>
void gtk_box_pack_start (GtkBox *box, GtkWidget *child, gboolean expand,
gboolean fill, guint padding);
void gtk_box_pack_end (GtkBox *box, GtkWidget *child, gboolean expand,
gboolean fill, guint padding);
```

两个函数的返回：无。

在参数列表中，box 是一个指向框容器的指针，child 是需要添加的元件的指针，expand 是一个布尔值，如果设置为 TRUE，则表示为这个元件分配新的位置，fill 设置元件是否为填充框的所有区域，padding 表示元件之间的距离。可以在一个框中添加多个元件，不同的元件以添加的次序从左到右、从上到下(或者是从右到左，从下到上)排列。

程序 13.5 演示了垂直框的使用，程序首先在主窗口中添加了一个垂直框，垂直框的上面放置了一个标签，下面放置了一个文本框和一个按钮。标签上依次显示了由文本框中输入的字符信息，并对这些信息进行编号。该界面模拟了一些论坛网站上的用户发表评论时的操作界面。程序源代码如 vbox_example.c 所示。

【程序 13.5】使用垂直框：vbox_example.c。

```
#include <gtk/gtk.h>
GtkWidget *window; /*指向窗口的指针*/
GtkWidget *vbox; /*指向垂直框的指针*/
GtkWidget *table; /*指向表格的指针*/
GtkWidget *entry; /*指向文本框的指针*/
GtkWidget *button; /*指向按钮的指针*/
GtkWidget *label; /*指向标签的指针*/
char text[50];
```

```
char text1[50];
char a[50];
int i=0;

void on_clicked(GtkWidget *widget, gpointer data)   /*定义信号回调函数*/
{
    i++;
    gcvt ((float)i,3,text);
    strcat(text, ": ");
    strcpy(text1, gtk_entry_get_text (GTK_ENTRY (entry)));
      /*获取文本框的文本内容，并将其复制到 text 字符串数组中*/
    strcat(text, text1);
    strcat(a, "\n");
    strcat(a, text);        /*将新的字符串连接到字符串数组 a 的后面*/
    gtk_label_set_text (GTK_LABEL (label), a);
}

int main (int argc,char *argv[])
{
    gtk_init(&argc,&argv);
    window=gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window),
                          g_locale_to_utf8("我也来评论",-1,NULL,NULL,NULL));
                        /*为窗口设置标题，g_locale_to_utf8()函数支持中文字符显示*/
    gtk_container_border_width(GTK_CONTAINER(window),5);   /*设置窗口边框*/
    vbox = gtk_vbox_new (FALSE, 10);
    label=gtk_label_new (g_locale_to_utf8("评论内容",-1,NULL,NULL,NULL));/*新建标签*/
    table=gtk_table_new(2,2,FALSE);
    entry =gtk_entry_new ();
    button=gtk_button_new_with_label(g_locale_to_utf8("发表评论",-1,NULL,NULL,NULL));
    gtk_container_add (GTK_CONTAINER (window), vbox);
    gtk_box_pack_start (GTK_BOX (vbox), label, TRUE, FALSE, 5);
    gtk_box_pack_start (GTK_BOX (vbox), table, TRUE, FALSE, 5);
    gtk_table_attach (GTK_TABLE(table), entry, 0, 2, 0, 1,
                      (GtkAttachOptions)(GTK_EXPAND), (GtkAttachOptions)(0), 5, 5);
    gtk_table_attach (GTK_TABLE(table), button, 1, 2, 1, 2,
                      (GtkAttachOptions)(0), (GtkAttachOptions)(0), 0, 0);
    g_signal_connect (G_OBJECT(button), "clicked", G_CALLBACK(on_clicked), window);
    gtk_widget_show_all (window); /*显示窗口中的所有元件*/
    gtk_main();
    return 0;
}
```

使用 gcc 编译 vbox_example.c，生成可执行文件 vbox_example，命令形式如下：

```
#gcc –o vbox_example vbox_example.c `pkg-config --libs --cflags gtk+-2.0`
```

运行程序，得到的输出结果如图 13.7 所示。

```
# ./vbox_example
```

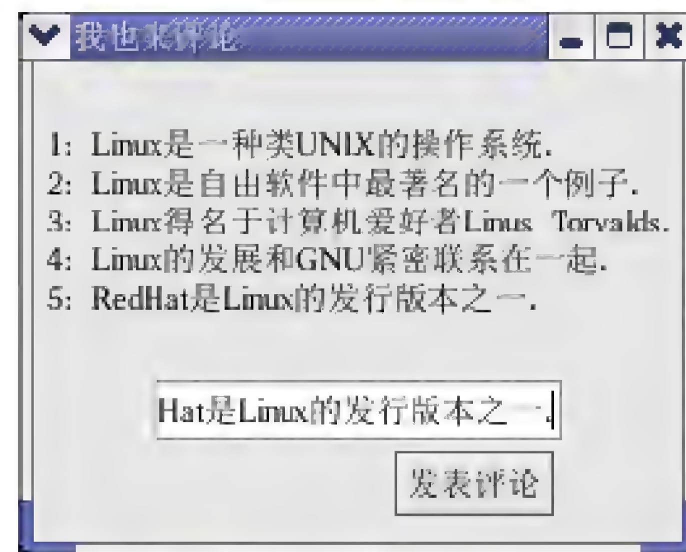


图 13.7 垂直框

从中可以看到，垂直框中的 3 个元件(标签、文本框、按钮)从上到下依次排列，并且每个放入框中的元件并不是同样的高或宽，而是根据元件自身的大小进行自动调整。

13.3.3 窗格

窗格也是 GTK+图形界面编程中常用的布局方式之一，它可以把一个界面划分成水平或垂直的两个区域，拖动两个区域的分界线，可以改变两个窗格的大小。窗格有水平窗格和垂直窗格两种。

1. 创建窗格

水平窗格使用函数 gtk_hpaned_new()生成，而垂直窗格使用函数 gtk_vpaned_new()生成。它们的函数原型如下：

```
#include <gtk/gtk.h>
GtkWidget *gtk_hpaned_new (void);
GtkWidget *gtk_vpaned_new (void);
```

两个函数的返回：若成功，则返回一个 GtkWidget 类型的指针；若失败，则返回空指针 NULL。

从函数的原型可以看到，这两个函数都没有参数，返回值是一个表示窗格的指针。同样地，建立一个窗格后，需要调用 gtk_container_add()函数将这个窗格添加到窗口中，并且需要调用显示函数 gtk_widget_show()来显示这个窗格。

可以调用 gtk_paned_set_position()函数设置窗格第一个区域的大小，函数原型如下：

```
#include <gtk/gtk.h>
void gtk_paned_set_position (GtkPaned *paned, gint position);
```

返回：无。

参数 paned 是一个表示窗格的指针，position 是一个整型数，表示这个窗格的分界线到边界的位置。

2. 在窗格中添加元件

作为 GTK+的容器，在没有向窗格中添加任何元件时，窗格是不能显示的(如同前面介绍的表格和框)。把元件添加到窗格中的函数调用是 gtk_paned_pack1()和 gtk_paned_pack2()，它们的函数原型如下：

```
#include <gtk/gtk.h>
void gtk_paned_pack1 (GtkPaned *paned, GtkWidget *child,
gboolean resize, gboolean shrink);
void gtk_paned_pack2 (GtkPaned *paned, GtkWidget *child,
```

```
gboolean resize, gboolean shrink);
```

返回：无。

这两个函数的使用方法是相同的。gtk_paned_pack1 的作用是把元件添加到窗格的第一个区域中，gtk_paned_pack2 的作用是把元件添加到窗格的第二个区域中。参数 paned 表示作为容器的窗格，child 表示将要装入的元件，resize 是一个布尔值，如果设置为 TRUE，则表示当窗格的大小改变时装入的元件随之改变大小，反之亦反。shrink 设置为 TRUE 时，表示如果窗格比这个元件小，元件会自动改变大小，反之亦反。

程序 13.6 创建了一个水平窗格，并在窗格的左右两个区域分别添加了两个标签。程序源代码如 hpaned_example.c 所示。

【程序 13.6】使用水平窗格：hpaned_example.c。

```
#include <gtk/gtk.h>
int main (int argc,char *argv[])
{
    GtkWidget *window; /*指向窗口的指针*/
    GtkWidget *hpaned; /*指向水平窗格的指针*/
    GtkWidget *label1; /*指向标签 1 的指针*/
    GtkWidget *label2; /*指向标签 2 的指针*/
    gtk_init(&argc,&argv);
    window=gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window),
g_locale_to_utf8("Linux 与 Windws",-1,NULL,NULL,NULL));
    hpaned = gtk_hpaned_new();
    label1=gtk_label_new (g_locale_to_utf8("Linux 和 Windows 都是多用户操作系统
\nLinux 和 Windows 都支持多种文件系统
\nLinux 和 Windows 都支持多种物理设备端口
\nLinux 和 Windows 都支持多种网络协议",-1,NULL,NULL,NULL));/*新建标签 1*/
    label2=gtk_label_new (g_locale_to_utf8("Linux 的应用目标是网络而不是打印
\n 可选的 GUI
\nLinux 不使用文件名扩展来识别文件的类型
\n 所有的 Linux 命令和选项都区分大小写",-1,NULL,NULL,NULL));/*新建标签 2*/
    gtk_paned_pack1 (GTK_PANED (hpaned), label1, TRUE, TRUE);
    gtk_paned_pack2 (GTK_PANED (hpaned), label2, TRUE, TRUE);
    gtk_container_add (GTK_CONTAINER (window), hpaned);
    gtk_paned_set_position (GTK_PANED (hpaned), 40);
    gtk_widget_show_all (window); /*显示窗口中的所有元件*/
    gtk_main();
    return 0;
}
```

使用 gcc 编译 hpaned_example.c，生成可执行文件 hpaned_example，命令形式如下：

```
#gcc –o hpaned_example hpaned_example.c `pkg-config --libs --cflags gtk+-2.0`
```

运行程序，得到的输出结果如图 13.8 所示。

```
# ./hpaned_example
```

从程序的运行结果可以看到，两个标签被放置到窗格的左右两个区域，如图 13.8 所示。当拖动窗格的分界线时，标签会随之改变大小，如图 13.9 所示。

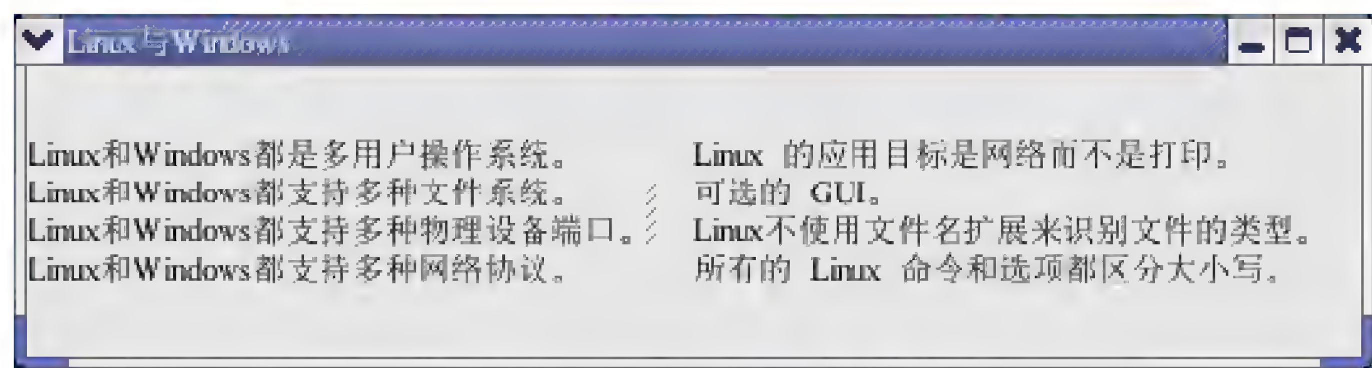


图 13.8　水平窗格

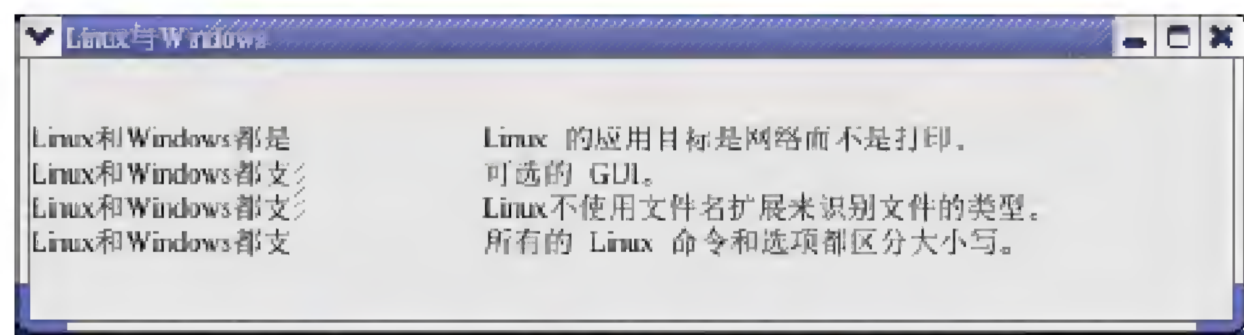


图 13.9　拖动分界线的效果

13.4 其他常用元件

GTK+的其他常用元件包括进度条、微调按钮、组合框、单选按钮、复选按钮、下拉菜单、对话框等，这些都是在平时的应用程序中常见的元件，本节向读者介绍创建这些元件的方法。

13.4.1　进度条、微调按钮、组合框

本小节介绍进度条、微调按钮和组合框的创建方法。

1. 进度条

进度条可以清晰地向用户显示某个应用程序的运行状态，是图形界面编程中常用的元件之一。创建进度条要使用 GtkAdjustment 元件。GtkAdjustment 用来存储初始值、上边界、下边界、步进值等信息。gtk_adjustment_new()函数用于创建 GtkAdjustment 元件，函数原型如下：

```
#include <gtk/gtk.h>
GtkObject *gtk_adjustment_new (gfloat value, gfloat lower, gfloat upper,
gfloat step_increment, gfloat page_increment, gloat page_size);
```

返回：若成功，则返回一个 GtkObject 类型的指针；若失败，则返回空指针 NULL。

函数中各个参数的含义如下：

- value：元件的初始值。
- lower：元件允许的最小值。
- upper：元件允许的最大值。
- step_increment：当鼠标左键按下时元件一次增加/减少的值。
- page_increment：当鼠标右键按下时元件一次增加/减少的值。
- page_size：不用。

GTK+中创建进度条的函数调用有两个，它们的函数原型如下：

```
#include <gtk/gtk.h>
GtkWidget *gtk_progress_bar_new (void);
GtkWidget *gtk_progress_bar_new_with_adjustment (GtkAdjustment *adjustment);
```

两个函数的返回：若成功，则返回一个 GtkWidget 类型的指针；若失败，则返回空指针NULL。

gtk_progress_bar_set_bar_style()函数用于设置进度条的样式，比如连续进度条和条块状的进度条，函数原型如下：

```
#include <gtk/gtk.h>
void gtk_progress_bar_set_bar_style (GtkProgressBar *pbar, GtkProgressBarStyle style);
```

返回：无。

参数 pbar 是一个指向进度条的指针，style 表示进度条的样式，取值如下：

- GTK_PROGRESS_CONTINUOUS：连续进度条。
- GTK_PROGRESS_DISCRETE：条块进度条。

gtk_progress_bar_set_orientation()函数用于设置进度条的方向，函数原型如下：

```
#include <gtk/gtk.h>
void gtk_progress_bar_set_orientation (GtkProgressBar *pbar,
GtkProgressBarOrientation orientation);
```

返回：无。

参数 pbar 是一个指向进度条的指针，orientation 表示进度条的方向，其取值如下：

- GTK_PROGRESS_LEFT_TO_RIGHT：从左往右显示进度。
- GTK_PROGRESS_RIGHT_TO_LEFT：从右往左显示进度。
- GTK_PROGRESS_BOTTOM_TO_TOP：从下往上显示进度。
- GTK_PROGRESS_TOP_TO_BOTTOM：从上往下显示进度。

gtk_progress_bar_update()函数用于更新进度条的进度状态，函数原型如下：

```
#include <gtk/gtk.h>
void gtk_progress_bar_update (GtkProgressBar *pbar, gfloat percentage);
```

返回：无。

参数 pbar 是一个指向进度条的指针，percentage 表示进度条的进度状态，即所占整个进度条的百分比。

2. 微调按钮

微调按钮通常用于让用户从一个取值范围里选择一个值，它由一个文本输入框和旁边的向上和向下两个按钮组成，单击某一个按钮会使文本框中的数值在一定范围内以某一个固定的步进值改变，文本框中也可以直接输入一个不超过上下边界的值。和进度条相同，创建微调按钮也要首先使用 GtkAdjustment 元件。

gtk_spin_button_new()函数用于创建一个微调按钮，函数原型如下：

```
#include <gtk/gtk.h>
GtkWidget* gtk_spin_button_new (GtkAdjustment *adjustment, gfloat climb_rate, gfloat digits);
```

返回：若成功，则返回一个 GtkWidget 类型的指针；若失败，则返回空指针 NULL。

参数 climb_rate 表示微调按钮每步的增加/减少值，digits 代表数值中包含的小数位数。

可以使用相关的函数调用获取和设置微调按钮的值。

gtk_spin_button_get_value_as_float()函数用于获取微调按钮的值(以 float 类型数值返回)，而 gtk_spin_button_set_value()函数用于设置微调按钮的值。它们的函数原型如下：

```
#include <gtk/gtk.h>
gfloat gtk_spin_button_get_value_as_float (GtkSpinButton *spin_button);
void gtk_spin_button_set_value (GtkSpinButton *spin_button, gfloat value);
```

参数 spin_button 是一个指向微调按钮的指针，参数 value 表示微调按钮的值。

3. 组合框

组合框是文本框和列表框的组合，通常也称之为下拉列表。创建组合框要使用 GList 元件，用于保存列表框中将要显示的字符串。g_list_append()函数用于向 GList 中添加字符串，它的函数原型如下：

```
#include <gtk/gtk.h>
void g_list_append (GList *glist, char *string);
```

返回：无。

参数 glist 是一个指向列表框的指针，string 指向将要添加至列表框中的字符串。

gtk_combo_new()函数用于创建一个组合框，函数原型如下：

```
#include <gtk/gtk.h>
GtkWidget *gtk_combo_new (void);
```

返回：若成功，则返回一个 GtkWidget 类型的指针；若失败，则返回空指针 NULL。

gtk_combo_set_popdown_strings()函数用于设置组合框中显示的字符串，函数原型如下：

```
#include <gtk/gtk.h>
void gtk_combo_set_popdown_strings (GtkCombo *combo, GList *strings);
```

返回：无。

参数 combo 是一个指向组合框的指针，strings 表示组合框中将要显示的字符串。

程序 13.7 是关于进度条、微调按钮和组合框的使用示例。程序中首先创建了一个垂直框，然后再在垂直框中依次添加了进度条、微调按钮和组合框等元件。程序源代码如 control_example.c 所示。

【程序 13.7】进度条、微调按钮和组合框的使用：control_example.c。

```
#include<gtk/gtk.h>
int main(int argc,char **argv)
{
    GtkWidget *window;
    GtkWidget *vbox;
    GtkWidget *label1;
    GtkWidget *label2;
```

```
    GtkWidget *label3;
    GtkObject *adjustment;
    GtkWidget *bar;
    GtkWidget *spinbutton;
    GList *glist;
    GtkWidget *combo;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window),
g_locale_to_utf8("控制元件",-1,NULL,NULL,NULL));
    gtk_container_border_width(GTK_CONTAINER(window),30);/*设置窗口边框的宽度 30*/
    vbox = gtk_vbox_new(FALSE,0);      /*创建一个垂直框*/
    gtk_container_add(GTK_CONTAINER(window),vbox);

    label1=gtk_label_new (g_locale_to_utf8("进度条：",-1,NULL,NULL,NULL));/*新建标签 1*/
    gtk_box_pack_start(GTK_BOX(vbox),label1,TRUE,TRUE,0); /*将标签 1 添加到垂直框*/
    /*下面是创建进度条*/
    adjustment = gtk_adjustment_new(70.0,0.0,100.0,1.0,0.0,0.0);
    bar = gtk_progress_bar_new_with_adjustment(GTK_ADJUSTMENT(adjustment));
    gtk_progress_bar_set_bar_style(GTK_PROGRESS_BAR(bar),
GTK_PROGRESS_CONTINUOUS);
    gtk_progress_bar_set_orientation(GTK_PROGRESS_BAR(bar),
    GTK_PROGRESS_LEFT_TO_RIGHT);
    gtk_box_pack_start(GTK_BOX(vbox),bar,TRUE,TRUE,15);/*将进度条添加到垂直框*/

    label2=gtk_label_new (g_locale_to_utf8("微调按钮：",-1,NULL,NULL,NULL));
/*新建标签 2*/
    gtk_box_pack_start(GTK_BOX(vbox),label2,TRUE,TRUE,0);/*将标签 2 添加到垂直框*/

    /*下面是创建微调按钮*/
    adjustment = gtk_adjustment_new(80.0,0.0,100.0,1.0,0.0,0.0);
    spinbutton = gtk_spin_button_new(GTK_ADJUSTMENT(adjustment),1.0,1);
    gtk_box_pack_start(GTK_BOX(vbox),spinbutton,TRUE,TRUE,15);
/*将微调按钮添加到垂直框*/

    label3=gtk_label_new (g_locale_to_utf8("下拉列表：",-1,NULL,NULL,NULL));
/*新建标签 3*/
    gtk_box_pack_start(GTK_BOX(vbox),label3,TRUE,TRUE,0);/*将标签 3 添加到垂直框*/

    /*下面是创建组合框*/
    glist = NULL;
    glist = g_list_append(glist,"banana");/*列表框中供选择的字符串*/
    glist = g_list_append(glist,"apple");
    glist = g_list_append(glist,"orange");
    glist = g_list_append(glist,"pear");
    glist = g_list_append(glist,"strawberry");
    combo = gtk_combo_new();
    gtk_combo_set_popdown_strings(GTK_COMBO(combo),glist);
    gtk_box_pack_start(GTK_BOX(vbox),combo,TRUE,TRUE,15);/*将组合框添加到垂直框*/
    gtk_widget_show_all (window);/*显示窗口中的所有元件*/
```

```
    gtk_main();
    return 0;
}
```

使用 gcc 编译 control_example.c，生成可执行文件 control_example，命令形式如下：

```
#gcc –o control_example control_example.c `pkg-config --libs --cflags gtk+-2.0`
```

运行程序，得到的输出结果如图 13.10 所示。

```
# ./control _example
```



图 13.10　进度条、微调按钮和组合框

读者可自行验证程序的运行结果，单击微调按钮和组合框的按钮，看看会发生什么情况。

另外，程序 13.7 中使用到了 gtk_container_border_width()函数，该函数用于设置窗口边框的宽度(如图 13.10 窗口边框宽度为 30)，是一个设置窗口属性的函数。

13.4.2　单选按钮、复选按钮

单选按钮(radio)和复选按钮(check)也是图形界面中常用的元件之一，它们提供一些可选的参数值供用户选择。单选按钮只能在众多的值当中选择一个，而复选按钮可以选择多个值。它们都有两种状态，一个是选中，另外一个是未选中。

创建复选按钮的函数有两个，函数原型如下：

```
#include<gtk/gtk.h>
GtkWidget *gtk_check_button_new (void);
GtkWidget *gtk_check _button_new_with_label (gchar *label);
```

两个函数的返回：若成功，则返回一个 GtkWidget 类型的指针；若失败，则返回空指针 NULL。

第一个函数创建一个无标签按钮，第二个函数创建带有文本标签的按钮，参数 label 指向按钮的标签文本。

创建单选按钮的函数也有两个，函数原型如下：

```
#include<gtk/gtk.h>
GtkWidget *gtk_radio_button_new(GSList *group);
GtkWidget *gtk_radio_button_new_with_label(GSList *group , gchar *label);
```

两个函数的返回：若成功，则返回一个 GtkWidget 类型的指针；若失败，则返回空指针 NULL。

同样地，第一个函数用于创建一个无标签按钮，第二个函数用于创建带有文本标签的按钮，参数 label 指向按钮的标签文本。由于 radio 按钮总是成组出现的，因此需要一个参数 group。

程序 13.8 创建了一个使用单选和复选按钮很常见的例子，程序首先定义了一个 4 行 4 列的表格，然后在表格的第 2 行和第 4 行分别放置 4 个复选按钮和 4 个单选按钮，在表格的第 1 行和第 3 行分别放置了一个标签。程序源代码如 check_radio.c 所示。

【程序 13.8】复选按钮和单选按钮的使用：check_radio.c。

```
#include<gtk/gtk.h>
int main (int argc,char *argv[])
{
    GtkWidget *window;
    GtkWidget *table;
    GtkWidget *label1;
    GtkWidget *label2;
    GSList *group;
    GtkWidget *check,*radio;
    gtk_init (&argc,&argv);
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window),
                             g_locale_to_utf8("单选、复选按钮",-1,NULL,NULL,NULL));
    gtk_container_border_width (GTK_CONTAINER(window),15);
    table=gtk_table_new(4,4,FALSE);
/*定义一个 4 行 4 列的表格，单元格大小会根据单元格中的元件大小自动调整*/
    label1=gtk_label_new (g_locale_to_utf8("业余时间活动爱好：",-1,NULL,NULL,NULL));
    label2=gtk_label_new (g_locale_to_utf8("平均每天学习时间：",-1,NULL,NULL,NULL));
    gtk_container_add(GTK_CONTAINER(window),table);
    gtk_table_attach (GTK_TABLE(table), label1, 0, 2, 0, 1,
                        (GtkAttachOptions)(0), (GtkAttachOptions)(0), 0, 0);
    gtk_table_attach (GTK_TABLE(table), label2, 0, 2, 2, 3,
                        (GtkAttachOptions)(0), (GtkAttachOptions)(0), 0, 5);

    /*以下生成 4 个 check 按钮，将它们分别加入到表格中*/
    check = gtk_check_button_new_with_label
(g_locale_to_utf8("篮球",-1,NULL,NULL,NULL));
    gtk_table_attach (GTK_TABLE(table), check, 0, 1, 1, 2,
                      (GtkAttachOptions)(0), (GtkAttachOptions)(0), 5, 10);
    check = gtk_check_button_new_with_label
(g_locale_to_utf8("足球",-1,NULL,NULL,NULL));
    gtk_table_attach (GTK_TABLE(table), check, 1, 2, 1, 2,
                      (GtkAttachOptions)(0), (GtkAttachOptions)(0), 5, 10);
    check = gtk_check_button_new_with_label
(g_locale_to_utf8("爬山",-1,NULL,NULL,NULL));
    gtk_table_attach (GTK_TABLE(table), check, 2, 3, 1, 2,
                      (GtkAttachOptions)(0), (GtkAttachOptions)(0), 5, 10);
    check = gtk_check_button_new_with_label
(g_locale_to_utf8("郊游",-1,NULL,NULL,NULL));
    gtk_table_attach (GTK_TABLE(table), check, 3, 4, 1, 2,
                      (GtkAttachOptions)(0), (GtkAttachOptions)(0), 5, 10);
 /*以下生成 4 个 radio 按钮，将它们加入到表格中。注意：生成第一个 radio 按钮时 group 参数为 NULL，
```

```
而后每次在该组中创建一个 radio 按钮都要使用 gtk_radio_button_group 函数获取新的 group 值*/
        radio = gtk_radio_button_new_with_label (NULL,
    g_locale_to_utf8("3-4 小时",-1,NULL,NULL,NULL));
        gtk_table_attach (GTK_TABLE(table), radio, 0, 1, 3, 4,
                        (GtkAttachOptions)(0), (GtkAttachOptions)(0), 5, 5);
        group = gtk_radio_button_group (GTK_RADIO_BUTTON(radio));
        radio = gtk_radio_button_new_with_label
     (group,g_locale_to_utf8("4-5 小时",-1,NULL,NULL,NULL));
        gtk_table_attach (GTK_TABLE(table), radio, 1, 2, 3, 4,
                        (GtkAttachOptions)(0), (GtkAttachOptions)(0), 5, 5);
        group = gtk_radio_button_group (GTK_RADIO_BUTTON(radio));
        radio = gtk_radio_button_new_with_label
                        (group,g_locale_to_utf8("5-6 小时",-1,NULL,NULL,NULL));
        gtk_table_attach (GTK_TABLE(table), radio, 2, 3, 3, 4,
                        (GtkAttachOptions)(0), (GtkAttachOptions)(0), 5, 5);
        group = gtk_radio_button_group (GTK_RADIO_BUTTON(radio));
        radio = gtk_radio_button_new_with_label
    (group,g_locale_to_utf8("6 小时以上",-1,NULL,NULL,NULL));
        gtk_table_attach (GTK_TABLE(table), radio, 3, 4, 3, 4,
                        (GtkAttachOptions)(0), (GtkAttachOptions)(0), 5, 5);
        gtk_widget_show_all (window);
        gtk_main();
        return 0;
    }
```

使用 gcc 编译 check_radio.c，生成可执行文件 check_radio，命令形式如下：

```
#gcc –o check_radio check_radio.c `pkg-config --libs --cflags gtk+-2.0`
```

运行程序，得到的输出结果如图 13.11 所示。

```
# ./ check_radio
```

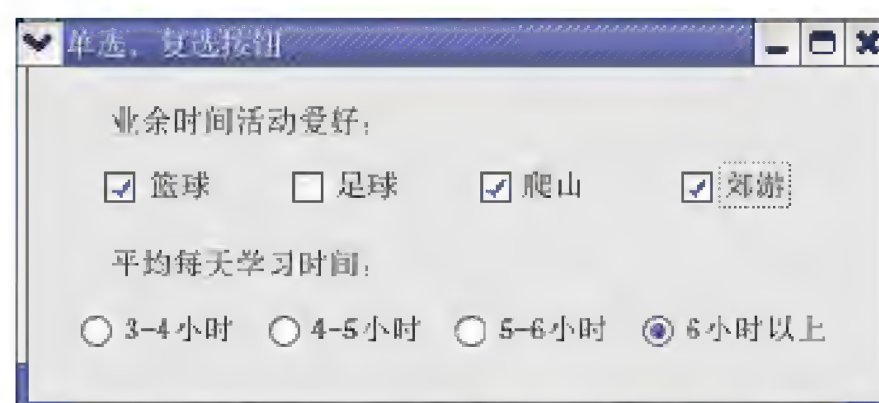


图 13.11　单选和复选按钮

对于图 13.11 所示的界面，是不是在问卷调查中很常见？可以看到，对于复选按钮，一次可以选择多个值，而对于单选按钮，一次只能选择其中的一个值。

13.4.3　下拉菜单

下面简单给出使用 GTK+库创建下拉菜单的相关函数调用，以及创建下拉菜单的步骤。然后读者可根据具体的实例来体会下拉菜单的生成方法。

创建菜单的函数调用有两个，函数原型如下：

```
#include<gtk/gtk.h>
```

```
GtkWidget *gtk_menu_new();
GtkWidget *gtk_menu_new_with_label(gchar *label);
```

两个函数的返回：若成功，则返回一个 GtkWidget 类型的指针；若失败，则返回空指针 NULL。

第一个函数创建一个无标签的菜单，第二个函数创建带有标签的菜单，参数 label 指向菜单的标签文本。

创建菜单项的函数调用类似于创建菜单(不再赘述)，函数原型如下：

```
#include<gtk/gtk.h>
GtkWidget *gtk_menu_item_new();
GtkWidget *gtk_menu_item_new_with_label(gchar *label);
```

插入菜单项的函数原型如下：

```
void gtk_menu_append (GtkMenu *menu, GtkWidget *child);
void gtk_menu_set_submenu (GtkMenuItem *item, GtkMenu *menu);
```

创建菜单条函数原型如下：

```
GtkWidget *gtk_menu_bar_new (void);
```

向菜单条中加入菜单：

```
void gtk_menu_bar_append (GtkMenuBar *bar, GtkWidget *child);
```

创建菜单的步骤如下：

- 使用 gtk_menu_new()或 gtk_menu_new_with_label()生成一个新菜单。
- 使用 gtk_menu_item_new()或 gtk_menu_new_with_label()生成一个新的菜单项，然后使用 gtk_menu_append()将菜单项加入到菜单中，使用 gtk_menu_item_new()或 gtk_menu_new_with_label()创建主菜单。
- 使用 gtk_menu_set_submenu()将各个菜单加入到主菜单中。
- 使用 gtk_menu_bar_new()创建菜单条。然后使用 gtk_menu_bar_append()把主菜单加入到菜单条上。

程序 13.9 演示了创建下拉菜单的详细过程。程序源代码如 menu_example.c 所示。

【程序 13.9】使用下拉菜单：menu_example.c。

```
#include<gtk/gtk.h>
int main(int argc,char *argv[])
{
    GtkWidget *window;
    GtkWidget *menu;
    GtkWidget *menubar;
    GtkWidget *rootmenu;
    GtkWidget *menuitem;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window),
    g_locale_to_utf8("下拉菜单",-1,NULL,NULL,NULL));
    gtk_widget_set_usize (window, 200, 40);
```

```
    /*下面是创建一个新菜单，然后创建 3 个菜单项，并把这 3 个菜单项加入到菜单中*/
    menu = gtk_menu_new();
    menuitem = gtk_menu_item_new_with_label
(g_locale_to_utf8("新建",-1,NULL,NULL,NULL));
    gtk_menu_append (GTK_MENU(menu),menuitem);
    menuitem = gtk_menu_item_new_with_label
(g_locale_to_utf8("打开",-1,NULL,NULL,NULL));
    gtk_menu_append (GTK_MENU(menu),menuitem);
    menuitem = gtk_menu_item_new_with_label
(g_locale_to_utf8("关闭",-1,NULL,NULL,NULL));
    gtk_menu_append (GTK_MENU(menu),menuitem);
    menuitem = gtk_menu_item_new_with_label
(g_locale_to_utf8("保存",-1,NULL,NULL,NULL));
    gtk_menu_append (GTK_MENU(menu),menuitem);
    rootmenu = gtk_menu_item_new_with_label
(g_locale_to_utf8("文件  ",-1,NULL,NULL,NULL)); /*创建一个主菜单*/
    gtk_menu_item_set_submenu(GTK_MENU_ITEM(rootmenu),menu);
                                        /*将菜单加入到主菜单中*/
    menubar = gtk_menu_bar_new();      /*创建菜单条*/
    gtk_menu_bar_append(GTK_MENU_BAR(menubar),rootmenu);
    /*将主菜单条加入到菜单条中*/
    /*下面使用同样的方法，创建第二组菜单*/
    menu = gtk_menu_new();
    menuitem = gtk_menu_item_new_with_label
(g_locale_to_utf8("复制",-1,NULL,NULL,NULL));
    gtk_menu_append (GTK_MENU(menu),menuitem);
    menuitem = gtk_menu_item_new_with_label
(g_locale_to_utf8("剪切",-1,NULL,NULL,NULL));
    gtk_menu_append(GTK_MENU(menu),menuitem);
    menuitem = gtk_menu_item_new_with_label
(g_locale_to_utf8("粘贴",-1,NULL,NULL,NULL));
    gtk_menu_append(GTK_MENU(menu),menuitem);
    menuitem = gtk_menu_item_new_with_label
(g_locale_to_utf8("删除",-1,NULL,NULL,NULL));
    gtk_menu_append(GTK_MENU(menu),menuitem);
    rootmenu = gtk_menu_item_new_with_label
(g_locale_to_utf8("编辑",-1,NULL,NULL,NULL));
    gtk_menu_item_set_submenu(GTK_MENU_ITEM(rootmenu),menu);
    gtk_menu_bar_append(GTK_MENU_BAR(menubar),rootmenu);
    gtk_container_add(GTK_CONTAINER(window),menubar);   /*将菜单条加入到窗口中*/
    gtk_widget_show_all (window); /*显示窗口中的所有元件*/
    gtk_main();
    return 0;
}
```

使用 gcc 编译 menu_example.c，生成可执行文件 menu_example，命令形式如下：

```
#gcc –o menu_example menu_example.c `pkg-config --libs --cflags gtk+-2.0`
```

运行程序，得到的输出结果如图 13.12 所示。

```
# ./ menu_example
```

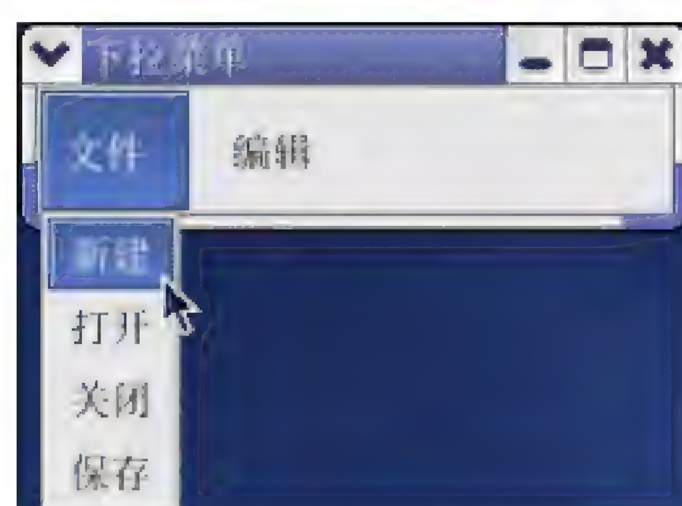


图 13.12　下拉菜单

如图 13.12 所示为鼠标单击“文件”选项时弹出的下拉菜单，读者可以为这些下拉菜单添加回调函数，比如单击“新建”标签选项时，便可以新建一个文本文件。下面向读者介绍 GTK+中的信号与回调函数。

13.5 信号与回调函数

回忆在 13.2.5 小节中的程序 13.3，当我们单击“提交”按钮时，文本框中的字符串信息会输出到终端，这表明发生了一个单击事件。实际上，单击按钮时系统自动调用了相关的事件响应函数，这就是本节将要给读者介绍的内容。

图形用户界面的程序是事件驱动的程序。程序进入 gtk_main()函数后，等待事件的发生，一旦发生某个事件，相应的信号将产生。如果程序中定义了相应的消息处理函数(或称回调函数)，系统会自动进行调用。

1. 添加信号

信号的作用是对某个元件添加一个用户交互的功能。函数 g_signal_connect()可以把一个信号处理函数(或称回调函数)添加到一个元件上，在元件和消息处理函数间建立关联，该函数的原型是：

```
#include <gtk/gtk.h>
gulong g_signal_connect (GtkObject *object, gchar *name
Gcallback callback_func, gpointer func_data);
```

返回：见下。

参数 object 是一个元件的指针，指向将产生信号的元件；name 表示消息或事件的类型，例如一个按钮所有的事件类型与含义如下：

- activate：激活的时候发生。
- clicked：单击以后发生。
- enter：鼠标指针进入这个按钮以后发生。
- leave：鼠标离开按钮以后发生。
- pressed：鼠标按下以后发生。
- released：鼠标释放以后发生。

参数 callback_func 表示信号产生后将要执行的回调函数，func_data 为传递给回调函数的数据，与 callback_func()函数(稍后介绍)的第二个参数相同。

该函数的返回值用于区分一个元件的一个事件对应的多个处理函数。一个元件上可以发生多个事件，比如单击一个按钮，双击一个按钮。对于一个元件上的每个事件可以有 0 个、1 个或多个处理函数。该事件发生时，将按声明的顺序逐个调用这些函数。对应于某个事件，如果元件没有定义处理函数(回调函数)，那么事件发生时将没有响应，系统忽略此事件。

在 g_signal_connect 函数中，使用事件的不同名称来区分不同的事件。例如，下面的代码表示了在鼠标离开按钮以后将要发生的动作。

```
g_signal_connect (G_OBJECT(button), "leave", G_CALLBACK(on_clicked), window);
```

2. 回调函数

消息处理函数(或称回调函数)的原型是：

```
#include <gtk/gtk.h>
void callback_func (GtkWidget *widget, gpointer func_data);
```

返回：无。

参数 widget 指向要接收消息的元件，参数 func_data 指向消息产生时传递给该函数的数据。

程序 13.10 演示了对一个按钮添加一个事件，主窗口中有一个按钮和一个标签，单击该按钮时产生信号，信号的回调函数的作用是由标签的文本来计数单击按钮的次数。程序源代码如 signal_example.c 所示。

【程序 13.10】单击按钮产生信号与事件：signal_example.c。

```
#include <gtk/gtk.h>
#include <stdlib.h>

int i=0;
GtkWidget *window; /*指向窗口的指针*/
GtkWidget *table; /*指向表格的指针*/
GtkWidget *label1; /*指向文本框的指针*/
GtkWidget *label2; /*指向文本框的指针*/
GtkWidget *label3; /*指向文本框的指针*/
GtkWidget *button; /*指向按钮的指针*/

void on_clicked(GtkWidget *widget, gpointer data)    /*定义信号回调函数*/
{
    char a[20];
    i++;
    gcvt ((float)i,3,a);
    gtk_label_set_text (GTK_LABEL (label2), a);      /*设置按钮的标签*/
}

int main (int argc,char *argv[])
{
    gtk_init(&argc,&argv);
```

```
    window=gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window),
                         g_locale_to_utf8("信号与事件",-1,NULL,NULL,NULL));
    gtk_container_border_width (GTK_CONTAINER(window),20);
    table=gtk_table_new(2,3,FALSE);
/*定义一个 2 行 3 列的表格，单元格大小会根据单元格中的元件大小自动调整*/
    label1=gtk_label_new (g_locale_to_utf8("单击按钮第",-1,NULL,NULL,NULL));
    label2=gtk_label_new (g_locale_to_utf8("0",-1,NULL,NULL,NULL));
    label3=gtk_label_new (g_locale_to_utf8("次",-1,NULL,NULL,NULL));
    button = gtk_button_new_with_label(g_locale_to_utf8("计数",-1,NULL,NULL,NULL));
/* "计数" 按钮*/
    gtk_container_add (GTK_CONTAINER (window), table); /*将表格添加到窗口中*/
    gtk_table_attach (GTK_TABLE(table), label1, 0, 1, 0, 1,
                    (GtkAttachOptions)(0), (GtkAttachOptions)(0), 0, 10);
    gtk_table_attach (GTK_TABLE(table), label2, 1, 2, 0, 1,
                    (GtkAttachOptions)(0), (GtkAttachOptions)(0), 0, 10);
    gtk_table_attach (GTK_TABLE(table), label3, 2, 3, 0, 1,
                    (GtkAttachOptions)(0), (GtkAttachOptions)(0), 0, 10);
    gtk_table_attach (GTK_TABLE(table), button, 2, 3, 1, 2,
                    (GtkAttachOptions)(0), (GtkAttachOptions)(0), 0, 10);
    g_signal_connect (G_OBJECT(button), "clicked", G_CALLBACK(on_clicked), window);
    gtk_widget_show_all (window); /*显示窗口*/
    gtk_main();
    return 0;
}
```

使用 gcc 编译 signal_example.c，生成可执行文件 signal_example，命令形式如下：

```
#gcc –o signal_example signal_example.c `pkg-config --libs --cflags gtk+-2.0`
```

运行程序，得到的输出结果如图 13.13 所示。

```
#./signal_example
```

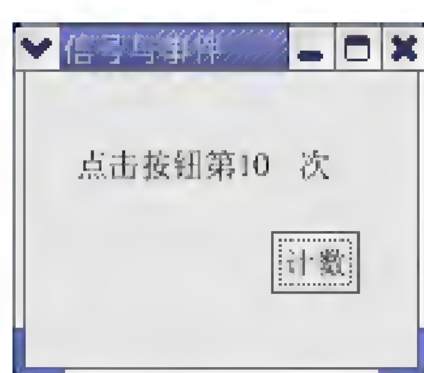


图 13.13　标题显示单击按钮的次数

如图 13.13 所示为单击“计数”按钮 10 次后的结果。可见，单击按钮时的确产生了相应的信号，并由系统自动调用了回调函数。

13.6 本章小结

本章向读者介绍了 Linux 下的图形界面编程——GTK+库，主要讲解了 GTK+库中的界面

基本元件(窗口、标签、按钮和文本框)，以及界面布局元件(表格、框和窗格)和其他的常用元件(进度条、微调按钮、组合框、单选按钮、复选按钮、下拉菜单、对话框等)，最后以一个实例介绍了 GTK+信号与回调函数。通过本章的学习，读者应该掌握基于 GTK+库的图形界面编程技术，并能设计出较复杂的界面。

实战演练

1. 编写一个程序，创建一个简单的 GTK+窗口界面，窗口的标题设置为“Linux Gtk+ programs”，窗口的大小为宽度 200，高度 150，窗口的左边距为 300，上边距也为 300，程序运行结果如图 13.4 所示。

2. 在窗口中添加一个标签，并将标签的文本设置为“Linux Gtk+ label”，程序运行结果如图 13.15 所示。

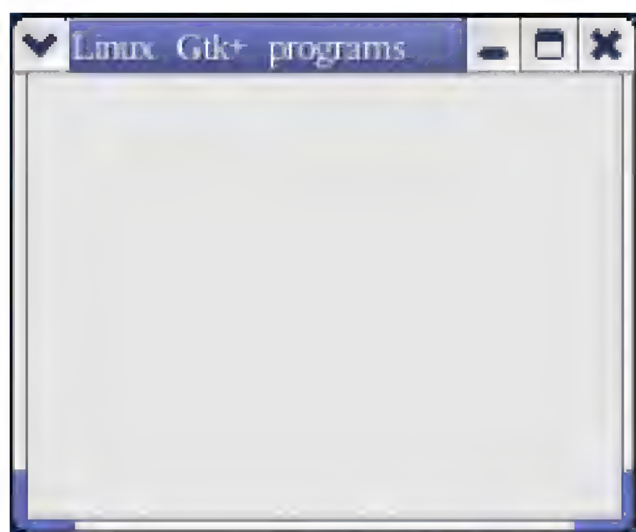


图 13.14　第 1 题图



图 13.15　第 2 题图

3. 编写一个程序，在窗口中添加一个文本框，要求该文本框中最多可输入 30 个字符，程序运行结果如图 13.16 所示。

4. 编写一个程序，创建一个 2 行 2 列的表格，并合并第 2 列的两个单元格，在表格中分别添加一个按钮、一个文本框和一个标签，程序运行结果如图 13.17 所示。

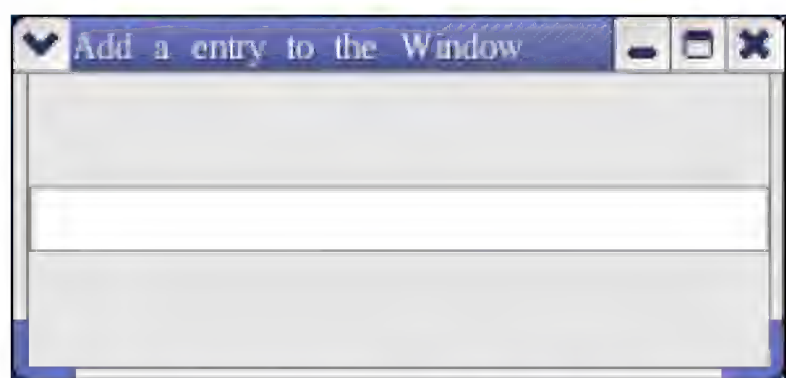


图 13.16　第 3 题图

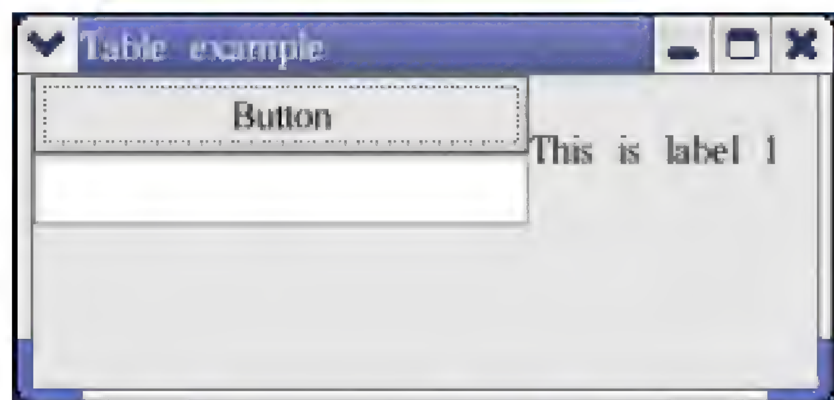


图 13.17　第 4 题图

5. 编写一个程序，创建一个垂直框，并在该垂直框中依次添加标签、按钮、文本框 3 个元件，使 3 个元件平均分布在窗口界面中，程序运行结果如图 13.18 所示。

6. 编写一个程序，创建一个微调按钮，使按钮的初始值为 1.0，最大值为 10，微调步进值为 0.1，程序运行结果如图 13.19 所示。

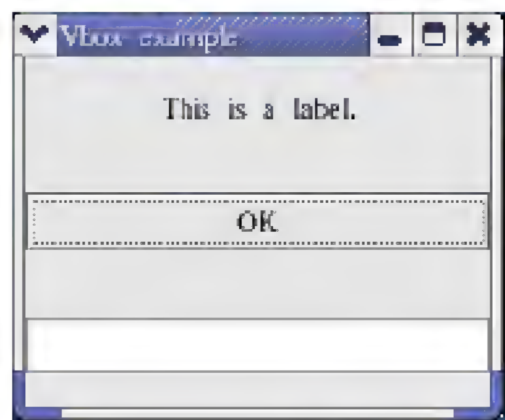


图 13.18　第 5 题图

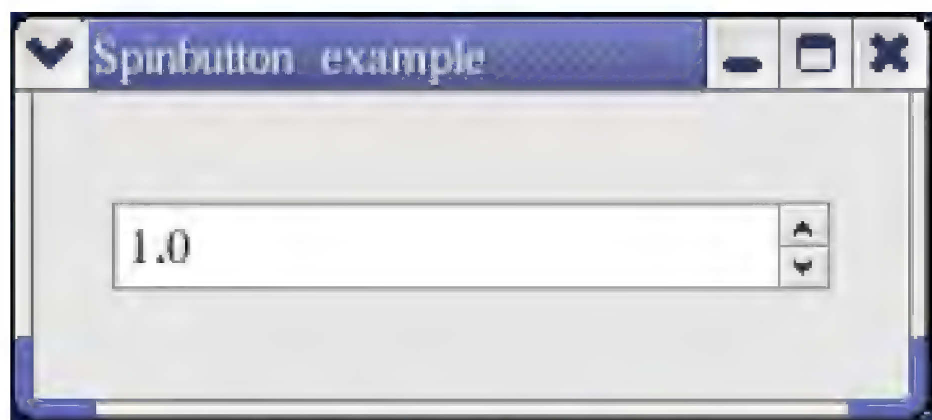


图 13.19　第 6 题图

7. 编写一个程序，在窗口界面中生成 3 个单选按钮和 2 个复选按钮，3 个单选按钮的选项分别为 Chinese、Math 和 English，2 个复选按钮的选项为 Teacher 和 Student，程序运行结果如图 13.20 所示。

8. 编写一个程序，在窗口中添加一个按钮，单击按钮时在按钮的标签上显示用户单击的次数。如图 13.21 所示为单击 3 次按钮后的效果图。

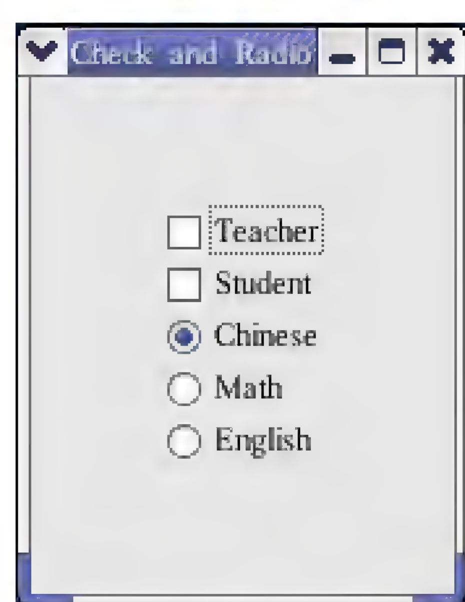


图 13.20　第 7 题图

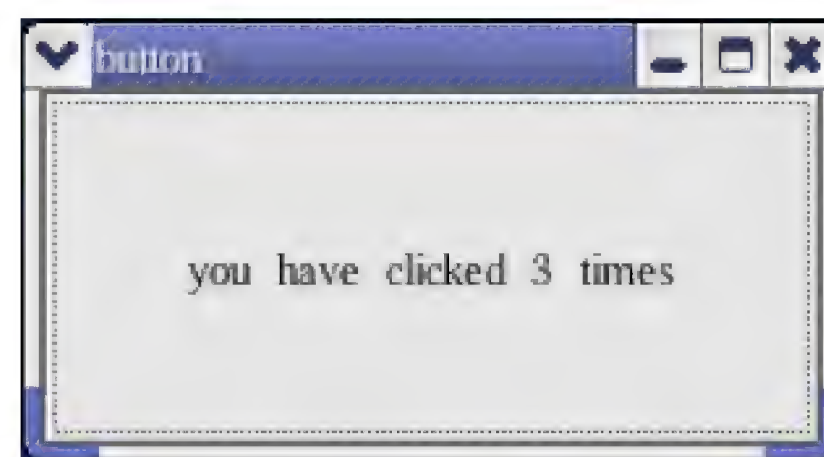


图 13.21　第 8 题图

第III部分

实 战 篇



第14章

设计Linux下的计算器

熟悉 Windows 的用户都知道，Windows 操作系统自带了一个计算器软件，用户可以在 DOS 命令行运行命令“c:\windows\system32\calc.exe”，或是从“开始”→“程序”→“附件”→“计算器”中打开它。而在 Linux 下，虽然有几个系统自带的命令可以完成简单的计算功能(比如“bc”和“expr”命令)，但对于习惯使用图形界面的用户来说，想使用类似于 Windows 下的图形化的计算器是不可能的。本章将带领读者自行设计一个 Linux 下的计算器，弥补 Linux 操作系统这一微小的缺陷。



本章内容：

◎ 软件功能分析
◎ 程序模块的划分
◎ 软件的具体实现
◎ 软件使用效果演示

14.1 软件功能分析

计算器程序主要包括界面、用户输入、数据运算、结果输出 4 个方面的内容。本节将分析该计算器程序的功能与要求。在编写复杂程序以前，需要对这个程序的功能进行认真分析，将复杂的功能拆解成多个简单模块。

首先，本章要向读者讲解的计算器界面是基于 GTK+图形界面编程库的实现。程序的界面需要用下面这些方法来实现：

- 需要有一个窗口，用来容纳所有的元件。
- 需要有一个文本框，用来接受和显示用户的数据输入，以及运算结果的输出显示。
- 需要有 4 个单选按钮，分别为十六进制、十进制、八进制和二进制。通过选择这些按钮来决定用户输入的数据及进行运算的进制法则，程序开启时默认为十进制。
- 需要有 0～9、A～F、pi(表示圆周率Π)一共 17 个数字按钮(A～F 用于十六进制)，用户可以单击这些按钮实现数据的输入。
- 需要有一个小数点按钮，实现小数点的输入。
- 需要有一个正负选择按钮，用户可以单击这个按钮选择输入的数据为正数，还是负数。
- 需要有 +、-、×、÷4 种基本运算的功能按钮，实现 4 种运算的输入。
- 其他运算功能的按钮，包括 sin、cos、tan、Exp、x^y、x^3、x^2、ln、log、n!、1/x、And、Not、Or、Xor、Mod(求模)、Int(取整)等。
- 需要有一个复位清零(CR)按钮，用来清除用户的输入。
- 需要有一个等于号按钮，完成数据的计算，并将结果显示出来。

程序的主要输入方式是单击按钮输入数据，这个交互需要实现下面的功能：

- 用户单击数字按钮时，将数字添加到文本框中。
- 用户单击正负选择按钮时，如果文本框中已有的数字为正数，则变成负数；若已有的数字为负数，则变成正数。并且，在输入数字之后，才能再输入正或负符号。
- 用户单击小数点按钮时，如果没有小数点，则在文本框中添加一个小数点；如果已有小数点，则不添加。
- 用户单击运算符号按钮时，需要记录数据的运算方法。
- 用户单击等于号按钮时，需要完成数据的计算，并显示运算结果。
- 用户单击清零(CR)按钮时，清除用户输入的数据，并将运算法则默认为加法。

处理数据是程序运算功能的最终实现模块。运算数据模块需要完成下面的功能：

- 用户单击运算符号按钮时，需要有一个变量来存储运算的方法。
- 用户完成一个数字的输入时，需要有一个变量来存储这个输入的数据。
- 从文本框中取得的数据是一个字符串，需要将这个字符串转化成浮点型数。
- 数据运算的结果是浮点型变量，需要转换成字符串以后设置成文本框的文本。
- 当用户选择不同的进制单选按钮时，需要有一个变量来存储当前的进制法则，以显示不同的界面，以及对输入、运算、输出数据的不同处理。

另外，在程序的运算部分，需要对用户的错误输入产生告警提示信息，包括下面几种情形：

- 除数不能为零。当用户输入 0 作为除数时(比如 1/x 求导、除法、取模运算等情况)，此时文本框显示中文“除数不能为零”，提示用户重新输入。
- 对数必须为正数。当用户输入负数或 0 作为对数时(比如 ln、log 运算)，文本框显示中文“对数必须为正数”，提示用户重新输入。
- 对于阶乘运算(n!)，用户输入的数值 n 必须为非负数。当输入的数值小于零时，在文本框显示中文“函数输入无效”，提示用户重新输入。

整个计算器程序运行稳定可靠，实现了同 Windows 下的计算器十分相近的功能，甚至比 Windows 的计算器更为完美，比如它(将要带领读者设计的计算器)可以实现十六进制、八进制和二进制的小数运算功能，并支持它们的负数符号的输入及负数运算功能等。

14.2 程序模块的划分

上一节分析了计算器软件程序需要完成的功能，这些功能是通过不同的函数来实现的。将一个较大型的程序划分为多个子程序和模块，有助于软件的实现和调试。本节向读者介绍计算器程序的函数和功能模块划分。

(1) 头文件与全局变量的定义。程序需要在不同函数里对各个元件进行访问，这些访问是通过全局变量来实现的。在程序最开始部分需要定义程序的全局变量。

(2) 用户单击单选按钮时，可以选择不同的计算器界面，并按照不同的进制法则进行计算，包括十六进制、十进制、八进制和二进制。它们各自的界面显示函数的定义方法如下所示：

```
void show_Hex_window();        /*十六进制界面显示函数*/
void show_Dec_window();        /*十进制界面显示函数*/
void show_Oct_window();        /*八进制界面显示函数*/
void show_Bin_window();        /*二进制界面显示函数*/
```

(3) 各进制间的转换函数。在不同的进制界面下，用户单击数字按钮输入的数值是按照相应的进制法则进行存储的，但在运算时都需要转换成十进制数进行计算(因为我们的编译器默认是只识别十进制数的)，计算结束后需将结果转换成十六进制、八进制或者二进制数值所对应的字符串，最后将该字符串添加在文本框中。进制间转换函数的定义方法如下所示：

```
int Conversion (char num[20], int t, int n);
```

该函数返回一个整型数，转换成功时返回 0，否则返回-1。参数列表中，num 表示从文本框中取得的字符串数值，整数 t 表示这个字符串代表的数值的进制数，整数 n 表示将要转换成的进制数。

(4) 用户单击按钮输入数字函数。用户单击数字按钮时，需要将这个按钮上的数字(即按钮的标签)添加到文本框的后面。有两个数字输入函数，它们的定义方法分别如下所示：

```
void input (GtkWidget *widget, gpointer data);
void input_pi (GtkWidget *widget, gpointer data);
```

第一个函数用于数字 0～9、A～F 的输入，第二个函数用于 pi(圆周率Π)的输入，当用户

单击这个按钮时，在文本框显示“3.14159265358979323846264338327 95”字符串。

(5) 正负选择函数。用户单击正负按钮时，会将最近一次输入的数字置为正或负。当文本框中的数字为正时，单击按钮使数字变为负；当文本框中的数字为负时，单击按钮使数字变为正数。函数的定义方法如下：

```
void Sign (GtkWidget *widget, gpointer data);
```

(6) 小数点输入函数。当用户单击小数点按钮时，这个函数对事件进行处理：

```
void dot (GtkWidget *widget, gpointer data);
```

(7) 各种运算功能按钮输入函数。用户单击这些运算按钮时，函数首先将运算方法存储在全局变量 method 中，然后调用相应的运算函数进行运算。下面给出这些函数的定义方法：

```
void Add (GtkWidget *widget, gpointer data);      /*加法运算*/
void Sub (GtkWidget *widget, gpointer data);      /*减法运算*/
void Mul (GtkWidget *widget, gpointer data);      /*乘法运算*/
void Division (GtkWidget *widget, gpointer data);   /*除法运算*/
void Mathpowxy (GtkWidget *widget,gpointer data); /*幂运算*/
void And (GtkWidget *widget,gpointer data);       /*逻辑与*/
void Or (GtkWidget *widget,gpointer data);        /*逻辑或*/
void Xor (GtkWidget *widget,gpointer data);       /*逻辑异或*/
void Mod (GtkWidget *widget,gpointer data);       /*模运算(取余)*/
void Sin (GtkWidget *widget,gpointer data);       /*求正弦(按弧度值)*/
void Cos (GtkWidget *widget,gpointer data);       /*求余弦(按弧度值)*/
void Tan (GtkWidget *widget,gpointer data);       /*求正切(按弧度值)*/
void Exp (GtkWidget *widget,gpointer data);       /*指数运算*/
void Cube (GtkWidget *widget,gpointer data);      /*立方*/
void Square (GtkWidget *widget,gpointer data);    /*平方*/
void Log_e (GtkWidget *widget,gpointer data);     /*底数为 e 求对数*/
void Log_10 (GtkWidget *widget,gpointer data);    /*底数为 10 求对数*/
void Factorial (GtkWidget *widget,gpointer data); /*阶乘*/
void Inverse (GtkWidget *widget,gpointer data);   /*求倒数*/
void Not (GtkWidget *widget,gpointer data);       /*逻辑非*/
void Floor (GtkWidget *widget,gpointer data);     /*取整*/
```

(8) 运算和输出函数。对于双目运算符，计算器必须记录两个数值后才能进行运算，用户单击“=”按钮时输出运算结果；而单目运算符则只需要一个数值，不需要单击“=”按钮，运算结果直接输出。另外，对于一些特殊的运算符，比如 1/x(求导)、/(除法)、Mod(取模)、ln、log(对数)、n!(阶乘)等，需要对用户输入的数值进行判断，非法输入时应给出相应的出错提示信息。这些功能由下面 3 个函数来完成：

```
void Binary_Operator ();       /*双目运算函数*/
void Right_output ();          /*单目运算的结果输出*/
void output ();                /*双目运算的结果输出*/
```

(9) 清除数据函数，这个函数可以清除所有已经输入的数据，程序回到初始状态，即全局变量 a 和 b(参与运算的两个数)被复位为 0，运算方法标识 method 同时也被复位为 0。函数定义方法如下：

```
void clear (GtkWidget *widget, gpointer data);
```

(10) 添加事件函数，作用是添加各个元件的交互事件：

```
void addsignal ();
```

(11) 单选按钮事件响应函数。用户单击单选按钮时，将显示不同的计算器界面，并使用全局变量 principle 来标识用户输入的数值。函数定义方法如下：

```
void on_clicked (GtkWidget *widget, gpointer data);
```

(12) 主函数，作用是完成所有模块和功能的调用：

```
int main (int argc, char *argv[]);
```

14.3 软件的具体实现

本节给出各个程序模块的源代码，包括头文件、十六进制界面显示函数、十进制界面显示函数、八进制界面显示函数、二进制界面显示函数、进制间转换函数、信号处理模块及主函数。其中，信号处理模块是整个计算器软件的核心模块。

14.3.1 头文件

在程序的最前面，需要包含程序的头文件。这里将程序需要的全局变量定义在头文件 myhead.h 中，包括两个参与运算的变量、小数点标识、运算符标识、进制标识，以及 GTK+图形界面编程中的常用元件指针。这些元件包括窗口、垂直框、表格、单选按钮、文本框和 42 个按钮。源代码如程序 14.1：myhead.h 所示。

【程序 14.1】自定义头文件：myhead.h。

```
#include <gtk/gtk.h>
double a,b;              /*定义两个参与运算的变量，双精度型*/
double p=0;
int hasdot;              /*是否有小数点*/
int method;              /*用于区别不同的运算*/
int principle;           /*标识不同的进制*/
char out[20]="0";
GtkWidget *window;       /*这一部分是定义元件*/
GtkWidget *vbox;         /*垂直框*/
GtkWidget *table1;       /*表格 1*/
GtkWidget *table2;       /*表格 2*/
GSList *group;
GtkWidget *radio;        /*单选按钮*/
GtkWidget *entry;        /*文本框*/
GtkWidget *button1;      /*42 个按钮*/
GtkWidget *button2;
GtkWidget *button3;
```

```
GtkWidget *button4;
GtkWidget *button5;
GtkWidget *button6;
GtkWidget *button7;
GtkWidget *button8;
GtkWidget *button9;
GtkWidget *button10;
GtkWidget *button11;
GtkWidget *button12;
GtkWidget *button13;
GtkWidget *button14;
GtkWidget *button15;
GtkWidget *button16;
GtkWidget *button17;
GtkWidget *button18;
GtkWidget *button19;
GtkWidget *button20;
GtkWidget *button21;
GtkWidget *button22;
GtkWidget *button23;
GtkWidget *button24;
GtkWidget *button25;
GtkWidget *button26;
GtkWidget *button27;
GtkWidget *button28;
GtkWidget *button29;
GtkWidget *button30;
GtkWidget *button31;
GtkWidget *button32;
GtkWidget *button33;
GtkWidget *button34;
GtkWidget *button35;
GtkWidget *button36;
GtkWidget *button37;
GtkWidget *button38;
GtkWidget *button39;
GtkWidget *button40;
GtkWidget *button41;
GtkWidget *button42;
```

14.3.2 十六进制界面显示函数

4 个单选按钮可以使用户选择不同的数值进制法则，并且计算器自动显示出不同的界面。比如在十六进制时，A、B、C、D、E、F 6 个数值按钮是需要显示的，而 pi(圆周率 Π)、sin、cos、tan 和 Exp 5 个按钮是不需要显示的，我们将该按钮的标签设置为“ ” (空字符串)，此时用户也不能操作这些按钮。十六进制界面显示函数的源代码如程序 14.2：show_Hex_window.c 所示。

【程序 14.2】十六进制界面显示函数：show_Hex_window.c。

```
#include <gtk/gtk.h>
void show_Hex_window()
{
    gtk_button_set_label (GTK_BUTTON (button1)," ");      /*按钮 pi 显示为空*/
    gtk_widget_show (button1);
    gtk_button_set_label (GTK_BUTTON (button2)," ");      /*按钮 sin 显示为空*/
    gtk_widget_show (button2);
    gtk_button_set_label (GTK_BUTTON (button3)," ");      /*按钮 cos 显示为空*/
    gtk_widget_show (button3);
    gtk_button_set_label (GTK_BUTTON (button4)," ");      /*按钮 tan 显示为空*/
    gtk_widget_show (button4);
    gtk_button_set_label (GTK_BUTTON (button6)," ");      /*按钮 Exp 显示为空*/
    gtk_widget_show (button6);
    gtk_button_set_label (GTK_BUTTON (button14),"7");     /*数字按钮 7 需要显示*/
    gtk_widget_show (button14);
    gtk_button_set_label (GTK_BUTTON (button15),"4");     /*数字按钮 4 需要显示*/
    gtk_widget_show (button15);
    gtk_button_set_label (GTK_BUTTON (button18),"A");     /*数字按钮 A 需要显示*/
    gtk_widget_show (button18);
    gtk_button_set_label (GTK_BUTTON (button19),"8");     /*数字按钮 8 需要显示*/
    gtk_widget_show (button19);
    gtk_button_set_label (GTK_BUTTON (button20),"5");     /*数字按钮 5 需要显示*/
    gtk_widget_show (button20);
    gtk_button_set_label (GTK_BUTTON (button21),"2");     /*数字按钮 2 需要显示*/
    gtk_widget_show (button21);
    gtk_button_set_label (GTK_BUTTON (button23),"B");     /*数字按钮 B 需要显示*/
    gtk_widget_show (button23);
    gtk_button_set_label (GTK_BUTTON (button24),"9");     /*数字按钮 9 需要显示*/
    gtk_widget_show (button24);
    gtk_button_set_label (GTK_BUTTON (button25),"6");     /*数字按钮 6 需要显示*/
    gtk_widget_show (button25);
    gtk_button_set_label (GTK_BUTTON (button26),"3");     /*数字按钮 3 需要显示*/
    gtk_widget_show (button26);
    gtk_button_set_label (GTK_BUTTON (button28),"C");     /*数字按钮 C 需要显示*/
    gtk_widget_show (button28);
    gtk_button_set_label (GTK_BUTTON (button33),"D");     /*数字按钮 D 需要显示*/
    gtk_widget_show (button33);
    gtk_button_set_label (GTK_BUTTON (button38),"E");     /*数字按钮 E 需要显示*/
    gtk_widget_show (button38);
    gtk_button_set_label (GTK_BUTTON (button42),"F");     /*数字按钮 F 需要显示*/
    gtk_widget_show (button42);
}
```

14.3.3 十进制界面显示函数

当单选按钮选中十进制时，计算器显示十进制界面。在十进制界面中，A、B、C、D、E、F 6 个数值按钮的标签被设置为“ ”(空字符串)，其他按钮均正常显示。源代码如程序 14.3：

show_Dec_window.c 所示。

【程序 14.3】十进制界面显示函数：show_Dec_window.c。

```
#include <gtk/gtk.h>
void show_Dec_window()
{
gtk_button_set_label (GTK_BUTTON (button1),"pi");      /*按钮 pi 需要显示*/
gtk_widget_show (button1);
gtk_button_set_label (GTK_BUTTON (button2),"sin");     /*按钮 sin 需要显示*/
gtk_widget_show (button2);
gtk_button_set_label (GTK_BUTTON (button3),"cos");     /*按钮 cos 需要显示*/
gtk_widget_show (button3);
gtk_button_set_label (GTK_BUTTON (button4),"tan");     /*按钮 tan 需要显示*/
gtk_widget_show (button4);
gtk_button_set_label (GTK_BUTTON (button6),"Exp");     /*按钮 Exp 需要显示*/
gtk_widget_show (button6);
gtk_button_set_label (GTK_BUTTON (button14),"7");      /*数字按钮 7 需要显示*/
gtk_widget_show (button14);
gtk_button_set_label (GTK_BUTTON (button15),"4");      /*数字按钮 4 需要显示*/
gtk_widget_show (button15);
gtk_button_set_label (GTK_BUTTON (button18)," ");      /*数字按钮 A 显示为空*/
gtk_widget_show (button18);
gtk_button_set_label (GTK_BUTTON (button19),"8");      /*数字按钮 8 需要显示*/
gtk_widget_show (button19);
gtk_button_set_label (GTK_BUTTON (button20),"5");      /*数字按钮 5 需要显示*/
gtk_widget_show (button20);
gtk_button_set_label (GTK_BUTTON (button21),"2");      /*数字按钮 2 需要显示*/
gtk_widget_show (button21);
gtk_button_set_label (GTK_BUTTON (button23)," ");      /*数字按钮 B 显示为空*/
gtk_widget_show (button23);
gtk_button_set_label (GTK_BUTTON (button24),"9");      /*数字按钮 9 需要显示*/
gtk_widget_show (button24);
gtk_button_set_label (GTK_BUTTON (button25),"6");      /*数字按钮 6 需要显示*/
gtk_widget_show (button25);
gtk_button_set_label (GTK_BUTTON (button26),"3");      /*数字按钮 3 需要显示*/
gtk_widget_show (button26);
gtk_button_set_label (GTK_BUTTON (button28)," ");      /*数字按钮 C 显示为空*/
gtk_widget_show (button28);
gtk_button_set_label (GTK_BUTTON (button33)," ");      /*数字按钮 D 显示为空*/
gtk_widget_show (button33);
gtk_button_set_label (GTK_BUTTON (button38)," ");      /*数字按钮 E 显示为空*/
gtk_widget_show (button38);
gtk_button_set_label (GTK_BUTTON (button42)," ");      /*数字按钮 F 显示为空*/
gtk_widget_show (button42);
}
```

14.3.4 八进制界面显示函数

当单选按钮选中八进制时，计算器显示八进制界面。在八进制界面中，pi(圆周率Π)、8、

9、A、B、C、D、E、F 9 个数值按钮，以及 sin、cos、tan 和 Exp 4 个功能按钮的标签被设置为“”(空字符串)，其他按钮均正常显示。源代码如程序 14.4：show_Oct_window.c 所示。

【程序 14.4】八进制界面显示函数：show_Oct_window.c。

```
#include <gtk/gtk.h>
void show_Oct_window()
{
gtk_button_set_label (GTK_BUTTON (button1)," ");      /*按钮 pi 显示为空*/
gtk_widget_show (button1);
gtk_button_set_label (GTK_BUTTON (button2)," ");      /*按钮 sin 显示为空*/
gtk_widget_show (button2);
gtk_button_set_label (GTK_BUTTON (button3)," ");      /*按钮 cos 显示为空*/
gtk_widget_show (button3);
gtk_button_set_label (GTK_BUTTON (button4)," ");      /*按钮 tan 显示为空*/
gtk_widget_show (button4);
gtk_button_set_label (GTK_BUTTON (button6)," ");      /*按钮 Exp 显示为空*/
gtk_widget_show (button6);
gtk_button_set_label (GTK_BUTTON (button14),"7");   /*数字按钮 7 需要显示*/
gtk_widget_show (button14);
gtk_button_set_label (GTK_BUTTON (button15),"4");   /*数字按钮 4 需要显示*/
gtk_widget_show (button15);
gtk_button_set_label (GTK_BUTTON (button18)," ");   /*数字按钮 A 显示为空*/
gtk_widget_show (button18);
gtk_button_set_label (GTK_BUTTON (button19)," ");   /*数字按钮 8 显示为空*/
gtk_widget_show (button19);
gtk_button_set_label (GTK_BUTTON (button20),"5");   /*数字按钮 5 需要显示*/
gtk_widget_show (button20);
gtk_button_set_label (GTK_BUTTON (button21),"2");   /*数字按钮 2 需要显示*/
gtk_widget_show (button21);
gtk_button_set_label (GTK_BUTTON (button23)," ");   /*数字按钮 B 显示为空*/
gtk_widget_show (button23);
gtk_button_set_label (GTK_BUTTON (button24)," ");   /*数字按钮 9 显示为空*/
gtk_widget_show (button24);
gtk_button_set_label (GTK_BUTTON (button25),"6");   /*数字按钮 6 需要显示*/
gtk_widget_show (button25);
gtk_button_set_label (GTK_BUTTON (button26),"3");   /*数字按钮 3 需要显示*/
gtk_widget_show (button26);
gtk_button_set_label (GTK_BUTTON (button28)," ");   /*数字按钮 C 显示为空*/
gtk_widget_show (button28);
gtk_button_set_label (GTK_BUTTON (button33)," ");   /*数字按钮 D 显示为空*/
gtk_widget_show (button33);
gtk_button_set_label (GTK_BUTTON (button38)," ");   /*数字按钮 E 显示为空*/
gtk_widget_show (button38);
gtk_button_set_label (GTK_BUTTON (button42)," ");   /*数字按钮 F 显示为空*/
gtk_widget_show (button42);
}
```

14.3.5 二进制界面显示函数

当单选按钮选中二进制时，计算器显示二进制界面。对于二进制界面，只显示 0 和 1 两个数值按钮，sin、cos、tan 和 Exp 4 个功能按钮依然不显示，其他按钮正常显示。源代码如程序 14.5：show_Bin_window.c 所示。

【程序 14.5】二进制界面显示函数：show_Bin_window.c。

```
#include <gtk/gtk.h>
void show_Bin_window()
{
gtk_button_set_label (GTK_BUTTON (button1)," ");        /*按钮 pi 显示为空*/
gtk_widget_show (button1);
gtk_button_set_label (GTK_BUTTON (button2)," ");       /*按钮 sin 显示为空*/
gtk_widget_show (button2);
gtk_button_set_label (GTK_BUTTON (button3)," ");       /*按钮 cos 显示为空*/
gtk_widget_show (button3);
gtk_button_set_label (GTK_BUTTON (button4)," ");       /*按钮 tan 显示为空*/
gtk_widget_show (button4);
gtk_button_set_label (GTK_BUTTON (button6)," ");       /*按钮 Exp 显示为空*/
gtk_widget_show (button6);
gtk_button_set_label (GTK_BUTTON (button14)," ");   /*数字按钮 7 显示为空*/
gtk_widget_show (button14);
gtk_button_set_label (GTK_BUTTON (button15)," ");   /*数字按钮 4 显示为空*/
gtk_widget_show (button15);
gtk_button_set_label (GTK_BUTTON (button18)," ");   /*数字按钮 A 显示为空*/
gtk_widget_show (button18);
gtk_button_set_label (GTK_BUTTON (button19)," ");   /*数字按钮 8 显示为空*/
gtk_widget_show (button19);
gtk_button_set_label (GTK_BUTTON (button20)," ");   /*数字按钮 5 显示为空*/
gtk_widget_show (button20);
gtk_button_set_label (GTK_BUTTON (button21)," ");   /*数字按钮 2 显示为空*/
gtk_widget_show (button21);
gtk_button_set_label (GTK_BUTTON (button23)," ");   /*数字按钮 B 显示为空*/
gtk_widget_show (button23);
gtk_button_set_label (GTK_BUTTON (button24)," ");   /*数字按钮 9 显示为空*/
gtk_widget_show (button24);
gtk_button_set_label (GTK_BUTTON (button25)," ");   /*数字按钮 6 显示为空*/
gtk_widget_show (button25);
gtk_button_set_label (GTK_BUTTON (button26)," ");   /*数字按钮 3 显示为空*/
gtk_widget_show (button26);
gtk_button_set_label (GTK_BUTTON (button28)," ");   /*数字按钮 C 显示为空*/
gtk_widget_show (button28);
gtk_button_set_label (GTK_BUTTON (button33)," ");   /*数字按钮 D 显示为空*/
gtk_widget_show (button33);
gtk_button_set_label (GTK_BUTTON (button38)," ");   /*数字按钮 E 显示为空*/
gtk_widget_show (button38);
gtk_button_set_label (GTK_BUTTON (button42)," ");   /*数字按钮 F 显示为空*/
gtk_widget_show (button42);
}
```

14.3.6 进制间转换函数

这一部分的函数实现了十六进制、十进制、八进制及二进制 4 种进制间的相互转换功能。在调用该函数的形参列表中，num 表示从文本框中取得的字符串数值，整数 t 表示这个字符串代表的数值的进制数，整数 n 表示将要转换成的进制数。比如下面的语句：

```
Conversion(num, 8, 10);
```

表示将 num 字符串数组所对应的一个八进制数转换成一个十进制数。

进制间的转换函数使用了两个全局变量，双精度浮点数 p 和字符数组 out。当其他进制转换成十进制时，将转换结果存储在浮点数 p 中；当十进制转换成其他进制时，将转换结果以字符串的形式存放在字符数组 out 中。

转换过程分为整数部分和小数部分，整数部分转换后需逆序输出，而小数部分顺序输出。事实上，程序 14.6 可以实现任意进制间的转换，比如四进制到十二进制间的转换，有兴趣的读者可以试一试。源代码如程序 14.6：Conversion.c 所示。

【程序 14.6】 进制间转换函数：Conversion.c。

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>

int Conversion(char num[20], int t, int n)
{
int i,ii,j,k,m,x,b[30],h[30],c;
double d,r;
ii=0;
p=0;    /*每次调用函数时，先将用于存储转换结果的全局变量 p 清零*/
memset (out, 0, 20);    /*将用于存储转换结果的全局变量数组 out 清零*/
m=strlen(num);   /*求字符串的长度*/
x=m;
for(k=0,j=0;k<m;k++)    /*对字符串进行分段，以小数点为界限，分为整数部分和小数部分*/
{
    if(num[k]=='.')
    {
        x=k;
        break;
    }
}

for(j=x-1;j>=0;j--)
{
    if(num[j]=='-') break;    /*符号的处理*/
    if(num[j]=='A') r=10;
    else if(num[j]=='B')
        r=11;
    else if(num[j]=='C')
        r=12;
```

```
        else if(num[j]=='D')
            r=13;
        else if(num[j]=='E')
            r=14;
        else if(num[j]=='F')
            r=15;
        else
            r=num[j]-'0';      /*将字符转换成数字*/
        p=p+r*(pow ((double)t,(double)(x-j-1)));   /*计算结果，整数部分*/
    }

    if(num[x]=='.')   /*小数部分的转换*/
    {
        for(j=x+1;j<m;j++)
        {
            if(num[j]=='A') r=10;
            else if(num[j]=='B')
                r=11;
            else if(num[j]=='C')
                r=12;
            else if(num[j]=='D')
                r=13;
            else if(num[j]=='E')
                r=14;
            else if(num[j]=='F')
                r=15;
            else
                r=num[j]-'0';   /*将字符转换成数字*/
            p=p+r*(pow ((double)t, (double)(x-j))); /*小数部分的转换*/
        }
    }
    if(n==10)   /*如果要转换成十进制*/
    {
        if(num[0]=='-')   /*如果字符串前面有负号，则数值取反，使其变为负数*/
        {
            p=-p;
        }
        return 0;   /*程序返回*/
    }

    else   /*如果要转换成其他进制，如十六进制、八进制、二进制*/
    {
        k=(int)p;
        i=0;
        while(k)     /*判定需要转换的数是否变为 0*/
        {
            h[i++]=k%n; /*取余数，进行进制转换，但是顺序与正确值相反*/
            k/=n;            /*转换一位之后进行相应的变化*/
        }
        c=0;
```

```
if(p!=(int)p)   /*选择性计算，如果是整数就不用进行这一步的计算了*/
{
    d=p-(int)p; /*取小数部分*/
    while(d!=0)
        {
        b[c]=(int)(d*n); /*算法为×N 取整*/
        d=d*n-b[c];
        c++;
        if(c>=10)
            break; /*主要是控制小数后面出现无限循环计算，跳出循环以免出现死循环*/
        }
}
if(num[0]=='-')   /*负数的情况*/
{
     out[0]='-';
     ii++;
}
for(j=i-1;j>=0;j--,ii++)   /*反序输出，大于 10 的数字进行相应的变化*/
{
        if(h[j]==10) out[ii]='A';
    else if(h[j]==11) out[ii]='B';
    else if(h[j]==12) out[ii]='C';
    else if(h[j]==13) out[ii]='D';
    else if(h[j]==14) out[ii]='E';
    else if(h[j]==15) out[ii]='F';
        else if(h[j]==9)    out[ii]='9';
        else if(h[j]==8)    out[ii]='8';
        else if(h[j]==7)    out[ii]='7';
        else if(h[j]==6)    out[ii]='6';
        else if(h[j]==5)    out[ii]='5';
        else if(h[j]==4)    out[ii]='4';
        else if(h[j]==3)    out[ii]='3';
        else if(h[j]==2)    out[ii]='2';
        else if(h[j]==1)    out[ii]='1';
        else out[ii]='0';
}
if(p!=(int)p)     /*选择性输出，这样可以节约输出时间和程序的运行时间*/
{
    out[ii++]='.';
    for(j=0;j<c;j++)   /*小数部分转换后的结果正序存储在全局字符数组变量 out 中*/
        {
        if(b[j]==10) out[ii]='A';
        else if(b[j]==11) out[ii]='B';
        else if(b[j]==12) out[ii]='C';
        else if(b[j]==13) out[ii]='D';
            else if(b[j]==14) out[ii]='E';
        else if(b[j]==15) out[ii]='F';
        else if(b[j]==9) out[ii]='9';
        else if(b[j]==8) out[ii]='8';
            else if(b[j]==7) out[ii]='7';
```

```
                else if(b[j]==6) out[ii]='6';
                else if(b[j]==5) out[ii]='5';
                else if(b[j]==4) out[ii]='4';
                    else if(b[j]==3) out[ii]='3';
                else if(b[j]==2) out[ii]='2';
                else if(b[j]==1) out[ii]='1';
                else out[ii]='0';
                    ii++;
                }
        }
        return 0;    /*程序返回*/
    }
}
```

14.3.7 信号处理模块

信号处理模块是整个计算器软件的核心部分，因为整个计算器的关键环节正是用户单击各个按钮时产生的信号，以及程序对这些信号的处理。

信号处理模块包括两个用户单击按钮输入数字函数、正负选择函数、小数点输入函数、各种运算功能按钮输入函数、运算输出函数、清除数据函数，以及添加事件函数。程序 14.7：Signal_Process.c 给出了信号处理模块中各个函数的源代码。

【程序 14.7】信号处理模块：Signal_Process.c。

```
#include <gtk/gtk.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void Binary_Operator()                /*双目运算*/
{
    char num[20];
    strcpy(num, gtk_entry_get_text(GTK_ENTRY(entry)));/*取得文本框的内容*/
    if(a==0)                          /*如果没有第一个数，则存储为第一个数*/
    {
        if(principle==16)
        {
            Conversion (num,16,10); /*将输入的十六进制数转换为十进制*/
             a=p;
        }
        if(principle==10)
        {
             a=atof(num);              /*直接转换成浮点型*/
        }
        if(principle==8)
        {
            Conversion (num,8,10);   /*将输入的八进制数转换为十进制*/
             a=p;
        }
```

```
            if(principle==2)
            {
                Conversion (num,2,10);  /*将输入的二进制数转换为十进制*/
                 a=p;
            }
          gtk_entry_set_text(GTK_ENTRY(entry),"");   /*清空文本框*/
    }
    else            /*如果已有第一个数，则应存储为第二个数*/
    {
          if(principle==16)
            {
                Conversion(num,16,10);  /*将输入的十六进制数转换为十进制*/
                 b=p;
            }
          if(principle==10)
            {
                b=atof(num);                /*直接转换成浮点型*/
            }
          if(principle==8)
            {
                Conversion(num,8,10);     /*将输入的八进制数转换为十进制*/
                 b=p;
            }
          if(principle==2)
            {
                Conversion(num,2,10);     /*将输入的二进制数转换为十进制*/
                 b=p;
            }
    }
    hasdot=0;    /*表示已经没有小数点*/
}

void Right_output()
{
    char num[20];
    gcvt(a,32,num);                     /*运算结果转换成字符串*/
    if(principle==16)
    {
            Conversion(num,10,16);   /*将运算结果(十进制字符串)转换成十六进制数*/
            gtk_entry_set_text(GTK_ENTRY(entry),out);       /*显示结果*/
    }
    if(principle==10)
    {
            gtk_entry_set_text(GTK_ENTRY(entry),num);       /*直接显示结果*/
    }
    if(principle==8)
    {
            Conversion(num,10,8);     /*将运算结果(十进制字符串)转换成八进制数*/
            gtk_entry_set_text(GTK_ENTRY(entry),out);       /*显示结果*/
    }
```

```
    if(principle==2)
    {
            Conversion(num,10,2);      /*将运算结果(十进制字符串)转换成二进制数*/
            gtk_entry_set_text(GTK_ENTRY(entry),out);        /*显示结果*/
    }
    a=0;
    b=0;
    method=0;
}

float fun(int c)     /*递归函数求阶乘*/
{
float d;
        if(c==0 || c==1) d=1;
        else d=fun(c-1)*c;
        return d;
}

void output()
{
  char num[20]="0";
  strcpy(num, gtk_entry_get_text(GTK_ENTRY(entry)));     /* 取得文本框输入的内容*/
  if(principle==16)
  {
       Conversion(num,16,10);   /*将输入的十六进制数转换为十进制*/
       b=p;
  }
  if(principle==10)
  {
       b=atof(num);                  /*直接转换成浮点型*/
  }
  if(principle==8)
  {
       Conversion(num,8,10);    /*将输入的八进制数转换为十进制*/
       b=p;
  }
  if(principle==2)
  {
       Conversion(num,2,10);    /*将输入的二进制数转换为十进制*/
       b=p;
  }
  switch(method)
  {
     case 0:
       a=a+b; Right_output(); break;
     case 1:
       a=a-b; Right_output(); break;
     case 2:
            a=a*b; Right_output(); break;
     case 3:
```

```
            if(b==0)
            {
            a=0; b=0; method=0;
                gtk_entry_set_text (GTK_ENTRY(entry),
g_locale_to_utf8("除数不能为零",-1,NULL,NULL,NULL)); /*显示出错信息*/
            }
        else
            {
            a=a/b;
                Right_output();
            }break;
          case 4:
        a=pow(a,b); Right_output(); break;
          case 5:
         a=((int)a) & ((int)b); Right_output(); break;
       case 6:
         a=((int)a) | ((int)b); Right_output(); break;
       case 7:
         a=((int)a) ^ ((int)b); Right_output(); break;
       case 8:
         if(b==0)
             {
           a=0; b=0; method=0;
               gtk_entry_set_text (GTK_ENTRY(entry),
                   g_locale_to_utf8("除数不能为零",-1,NULL,NULL,NULL)); /*显示出错信息*/
             }
          else
             {
            a=((int)a) % ((int)b);
                Right_output();
             }break;
           case 9:
          a=sin(b); Right_output(); break;
        case 10:
          a=cos(b); Right_output(); break;
        case 11:
              a=tan(b); Right_output(); break;
        case 12:
          a=exp(b); Right_output(); break;
           case 13:
          a=b*b*b; Right_output(); break;
           case 14:
          a=b*b; Right_output(); break;
        case 15:
              if(b<=0)
             {
            a=0; b=0; method=0;
                gtk_entry_set_text (GTK_ENTRY(entry),
                              g_locale_to_utf8("对数必须为正数",-1,NULL,NULL,NULL));
/*显示出错信息*/
```

```
                }
                else
                {
               a=log(b);
                    Right_output();
                }break;
          case 16:
                if(b<=0)
                {
               a=0; b=0; method=0;
                    gtk_entry_set_text (GTK_ENTRY(entry),
                                   g_locale_to_utf8("对数必须为正数",-1,NULL,NULL,NULL));
                                                       /*显示出错信息*/
                }
                else
                {
               a=log10(b);
                    Right_output();
                }break;
          case 17:
                if(b<0)
                {
               a=0; b=0; method=0;
                    gtk_entry_set_text (GTK_ENTRY(entry),
                                   g_locale_to_utf8("函数输入无效",-1,NULL,NULL,NULL));
/*显示出错信息*/
                }
             else
                {
               a=fun((int)(b));
                    Right_output();
                }break;
              case 18:
                if(b==0)
                {
              a=0; b=0; method=0;
                   gtk_entry_set_text (GTK_ENTRY(entry),
                                    g_locale_to_utf8("除数不能为零",-1,NULL,NULL,NULL));
/*显示出错信息*/
                }
                else
                {
               a=1/b;
                    Right_output();
                }break;
             case 19:
           a=~((int)b); Right_output(); break;
             case 20:
           a=floor(b); Right_output(); break;
        default: break;
```

```
    }
}

void Add(GtkWidget *widget, gpointer data)      /*加法运算*/
{
    method=0;
    Binary_Operator();
}

void Sub(GtkWidget *widget,gpointer data)       /*减法运算*/
{
    method=1;
    Binary_Operator();
}

void Mul(GtkWidget *widget,gpointer data)       /*乘法运算*/
{
    method=2;
    Binary_Operator();
}

void Division(GtkWidget *widget,gpointer data)     /*除法运算*/
{
    method=3;
    Binary_Operator();
}

void Mathpowxy(GtkWidget *widget,gpointer data)    /*幂运算*/
{
    method=4;
    Binary_Operator();
}

void And(GtkWidget *widget,gpointer data)      /*逻辑与*/
{
method=5;
    Binary_Operator();
}

void Or(GtkWidget *widget,gpointer data)       /*逻辑或*/
{
    method=6;
    Binary_Operator();
}

void Xor(GtkWidget *widget,gpointer data)      /*逻辑异或*/
{
    method=7;
    Binary_Operator();
}
```

```
void Mod(GtkWidget *widget,gpointer data)        /*模运算(取余)*/
{
    method=8;
    Binary_Operator();
}

void Sin(GtkWidget *widget,gpointer data)        /*模运算(取余)*/
{
    method=9;
    output();
}

void Cos(GtkWidget *widget,gpointer data)        /*模运算(取余)*/
{
    method=10;
    output();
}

void Tan(GtkWidget *widget,gpointer data)        /*模运算(取余)*/
{
    method=11;
    output();
}

void Exp(GtkWidget *widget,gpointer data)        /*模运算(取余)*/
{
    method=12;
    output();
}

void Cube(GtkWidget *widget,gpointer data)        /*模运算(取余)*/
{
    method=13;
    output();
}

void Square(GtkWidget *widget,gpointer data)        /*模运算(取余)*/
{
    method=14;
    output();
}

void Log_e(GtkWidget *widget,gpointer data)        /*模运算(取余)*/
{
    method=15;
    output();
}

void Log_10(GtkWidget *widget,gpointer data)        /*模运算(取余)*/
```

```
{
    method=16;
    output();
}

void Factorial(GtkWidget *widget,gpointer data)     /*模运算(取余)*/
{
    method=17;
    output();
}

void Inverse(GtkWidget *widget,gpointer data)     /*模运算(取余)*/
{
    method=18;
    output();
}

void Not(GtkWidget *widget,gpointer data)     /*模运算(取余)*/
{
    method=19;
    output();
}

void Floor(GtkWidget *widget,gpointer data)     /*模运算(取余)*/
{
    method=20;
    output();
}

void dot(GtkWidget *widget,gpointer data)
{
    if(hasdot==0) /*  没有小数点则添加一个小数点*/
    {
        gtk_entry_append_text (GTK_ENTRY (entry), gtk_button_get_label (widget));
        hasdot=1;/*   表示有一个小数点*/
    }
}

void Sign()
{
    char num[20];
    float c;
    strcpy(num, gtk_entry_get_text(GTK_ENTRY(entry)));/*取得文本框的内容*/
    c=atof(num);                    /*转换成浮点型*/
    c=-c;
    gcvt(c,32,num);                 /*结果转换成字符串*/
    gtk_entry_set_text(GTK_ENTRY(entry),num);     /*显示结果*/
}

void clear(GtkWidget *widget,gpointer data)
```

```
{
    gtk_entry_set_text(GTK_ENTRY(entry),"");
    hasdot=0;
    a=0;
    b=0;
    method=0;
}

void input (GtkWidget *widget, gpointer data)
{
    gtk_entry_append_text (GTK_ENTRY (entry), gtk_button_get_label(widget));
}

void input_pi (GtkWidget *widget, gpointer data)
{
    gtk_entry_set_text (GTK_ENTRY (entry), "3.14159265358979323846264338327950");
}

void addsignal()
{
/*下面的 17 个按钮实现数字的输入*/
g_signal_connect (G_OBJECT (button1), "clicked", G_CALLBACK (input_pi), NULL);
g_signal_connect (G_OBJECT (button14), "clicked", G_CALLBACK (input), NULL);
g_signal_connect (G_OBJECT (button15), "clicked", G_CALLBACK (input), NULL);
g_signal_connect (G_OBJECT (button16), "clicked", G_CALLBACK (input), NULL);
g_signal_connect (G_OBJECT (button17), "clicked", G_CALLBACK (input), NULL);
g_signal_connect (G_OBJECT (button18), "clicked", G_CALLBACK (input), NULL); /*A*/
g_signal_connect (G_OBJECT (button19), "clicked", G_CALLBACK (input), NULL);
g_signal_connect (G_OBJECT (button20), "clicked", G_CALLBACK (input), NULL);
g_signal_connect (G_OBJECT (button21), "clicked", G_CALLBACK (input), NULL);
g_signal_connect (G_OBJECT (button23), "clicked", G_CALLBACK (input), NULL); /*B*/
g_signal_connect (G_OBJECT (button24), "clicked", G_CALLBACK (input), NULL);
g_signal_connect (G_OBJECT (button25), "clicked", G_CALLBACK (input), NULL);
g_signal_connect (G_OBJECT (button26), "clicked", G_CALLBACK (input), NULL);
g_signal_connect (G_OBJECT (button28), "clicked", G_CALLBACK (input), NULL); /*C*/
g_signal_connect (G_OBJECT (button33), "clicked", G_CALLBACK (input), NULL); /*D*/
g_signal_connect (G_OBJECT (button38), "clicked", G_CALLBACK (input), NULL); /*E*/
g_signal_connect (G_OBJECT (button42), "clicked", G_CALLBACK (input), NULL); /*F*/
/*下面的按钮实现小数点的输入*/
g_signal_connect (G_OBJECT (button27), "clicked", G_CALLBACK (dot), NULL);
/*下面的按钮实现正负号的输入*/
g_signal_connect (G_OBJECT (button22), "clicked", G_CALLBACK (Sign), NULL);
/*下面的按钮实现各种运算的输入*/
g_signal_connect (G_OBJECT (button2), "clicked", G_CALLBACK (Sin), NULL);
g_signal_connect (G_OBJECT (button3), "clicked", G_CALLBACK (Cos), NULL);
g_signal_connect (G_OBJECT (button4), "clicked", G_CALLBACK (Tan), NULL);
g_signal_connect (G_OBJECT (button6), "clicked", G_CALLBACK (Exp), NULL);
g_signal_connect (G_OBJECT (button7), "clicked", G_CALLBACK (Mathpowxy), NULL);
g_signal_connect (G_OBJECT (button8), "clicked", G_CALLBACK (Cube), NULL);
g_signal_connect (G_OBJECT (button9), "clicked", G_CALLBACK (Square), NULL);
```

```
g_signal_connect (G_OBJECT (button10), "clicked", G_CALLBACK (Log_e), NULL);
g_signal_connect (G_OBJECT (button11), "clicked", G_CALLBACK (Log_10), NULL);
g_signal_connect (G_OBJECT (button12), "clicked", G_CALLBACK (Factorial), NULL);
g_signal_connect (G_OBJECT (button13), "clicked", G_CALLBACK (Inverse), NULL);
g_signal_connect (G_OBJECT (button32), "clicked", G_CALLBACK (Add), NULL);
g_signal_connect (G_OBJECT (button31), "clicked", G_CALLBACK (Sub), NULL);
g_signal_connect (G_OBJECT (button30), "clicked", G_CALLBACK (Mul), NULL);
g_signal_connect (G_OBJECT (button29), "clicked", G_CALLBACK (Division), NULL);
g_signal_connect (G_OBJECT (button35), "clicked", G_CALLBACK (And), NULL);
g_signal_connect (G_OBJECT (button36), "clicked", G_CALLBACK (Or), NULL);
g_signal_connect (G_OBJECT (button37), "clicked", G_CALLBACK (Mod), NULL);
g_signal_connect (G_OBJECT (button39), "clicked", G_CALLBACK (Not), NULL);
g_signal_connect (G_OBJECT (button40), "clicked", G_CALLBACK (Xor), NULL);
g_signal_connect (G_OBJECT (button41), "clicked", G_CALLBACK (Floor), NULL);
/* 下面的按钮实现复位功能*/
g_signal_connect (G_OBJECT (button34), "clicked", G_CALLBACK (clear), NULL);
/* 下面的按钮实现结果输出*/
g_signal_connect (G_OBJECT (button5), "clicked", G_CALLBACK (output), NULL);
g_signal_connect (G_OBJECT (window),"delete_event", gtk_main_quit, NULL);
}
```

14.3.8 主函数

主函数实现了计算器界面的初始化定义，以及各个模块和功能的调用。另外，单选按钮事件响应函数也在这部分给出。源代码如程序 14.8：main.c 所示。

【程序 14.8】主函数：main.c。

```
#include <stdlib.h>
#include <gtk/gtk.h>
#include <math.h>
#include "myhead.h"
#include "show_Hex_window.c"
#include "show_Dec_window.c"
#include "show_Oct_window.c"
#include "show_Bin_window.c"
#include "Signal_Process.c "
#include "Conversion.c"

void on_clicked(GtkWidget *widget, gpointer data)
{
    if(GTK_TOGGLE_BUTTON(widget)->active)
        {
            if((char *)data=="Hex")
            {
            show_Hex_window();
            principle=16;
            }
            if((char *)data=="Dec")
            {
```

```
            show_Dec_window();
                principle=10;
            }
            if((char *)data=="Oct")
            {
            show_Oct_window();
                principle=8;
            }
            if((char *)data=="Bin")
            {
            show_Bin_window();
                principle=2;
            }
        }
}

int main (int argc, char *argv[])
{
a=0;
b=0;    /*初始化两个参与运算的变量为零*/
hasdot=0;   /*小数点标识为 0，即默认不带小数点*/
gtk_set_locale();
gtk_rc_add_default_file("./gtkrc.zh_CN");     /*支持 GTK+的中文显示*/
gtk_init (&argc,&argv);           /*建立窗口*/
method=0;                         /*运算方法标识，初始默认为加法运算*/
window=gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_window_set_title (GTK_WINDOW (window),
                        g_locale_to_utf8("我的计算器",-1,NULL,NULL,NULL));

vbox = gtk_vbox_new (FALSE, 0); /*创建一个垂直框*/
gtk_container_add (GTK_CONTAINER (window), vbox);
gtk_widget_show (vbox);
table1 = gtk_table_new (2,4,FALSE);           /*建立一个 2 行 4 列的表格 1*/
gtk_box_pack_start (GTK_BOX (vbox), table1, TRUE, FALSE, 0);
gtk_widget_show (table1);
table2= gtk_table_new (5,9,FALSE);            /*建立一个 5 行 9 列的表格 2*/
gtk_box_pack_start (GTK_BOX (vbox), table2, TRUE, FALSE, 0);
gtk_widget_show (table2);
entry = gtk_entry_new ( );                    /*用于输入和输出的文本框*/
gtk_table_attach (GTK_TABLE (table1), entry, 0, 4, 0, 1,
                    (GtkAttachOptions) (GTK_EXPAND | GTK_FILL),
                    (GtkAttachOptions) (0),0,0);
gtk_widget_show (entry);
/*下面是 42 个按钮的定义和显示*/
button1 = gtk_button_new_with_mnemonic("pi");         /*pi*/
gtk_table_attach (GTK_TABLE (table2), button1, 0, 1, 0, 1,
                    (GtkAttachOptions) (GTK_FILL),
                    (GtkAttachOptions) (0), 0, 0);
gtk_widget_set_size_request (button1,40,30);
```

```
button2 = gtk_button_new_with_mnemonic("sin");            /*sin*/
gtk_table_attach (GTK_TABLE (table2), button2, 0, 1, 1, 2,
                  (GtkAttachOptions) (GTK_FILL),
                  (GtkAttachOptions) (0), 0, 0);

button3 = gtk_button_new_with_mnemonic("cos");            /*cos*/
gtk_table_attach (GTK_TABLE (table2), button3, 0, 1, 2, 3,
                  (GtkAttachOptions) (GTK_FILL),
                  (GtkAttachOptions) (0), 0, 0);

button4 = gtk_button_new_with_mnemonic("tan");            /*tan*/
gtk_table_attach (GTK_TABLE (table2), button4, 0, 1, 3, 4,
                  (GtkAttachOptions) (GTK_FILL),
                  (GtkAttachOptions) (0), 0, 0);

button5 = gtk_button_new_with_mnemonic("=");              /*==*/
gtk_widget_show (button5);/*常显示，不变*/
gtk_table_attach (GTK_TABLE (table2), button5, 0, 3, 4, 5,
                  (GtkAttachOptions) (GTK_FILL),
                  (GtkAttachOptions) (0), 0, 0);

button6 = gtk_button_new_with_mnemonic("Exp");            /*Exp*/
gtk_table_attach (GTK_TABLE (table2), button6, 1, 2, 0, 1,
                  (GtkAttachOptions) (GTK_FILL),
                  (GtkAttachOptions) (0), 0, 0);
gtk_widget_set_size_request (button6,40,30);

button7 = gtk_button_new_with_mnemonic("x^y");            /*x^y*/
gtk_widget_show (button7);/*常显示，不变*/
gtk_table_attach (GTK_TABLE (table2), button7, 1, 2, 1, 2,
                  (GtkAttachOptions) (GTK_FILL),
                  (GtkAttachOptions) (0), 0, 0);

button8 = gtk_button_new_with_mnemonic("x^3");            /*x^3*/
gtk_widget_show (button8);/*常显示，不变*/
gtk_table_attach (GTK_TABLE (table2), button8, 1, 2, 2, 3,
                  (GtkAttachOptions) (GTK_FILL),
                  (GtkAttachOptions) (0), 0, 0);

button9 = gtk_button_new_with_mnemonic("x^2");            /*x^2*/
gtk_widget_show (button9);/*常显示，不变*/
gtk_table_attach (GTK_TABLE (table2), button9, 1, 2, 3, 4,
                  (GtkAttachOptions) (GTK_FILL),
                  (GtkAttachOptions) (0), 0, 0);

button10 = gtk_button_new_with_mnemonic("ln");            /*ln*/
gtk_widget_show (button10);/*常显示，不变*/
gtk_table_attach (GTK_TABLE (table2), button10, 2, 3, 0, 1,
                  (GtkAttachOptions) (GTK_FILL),
                  (GtkAttachOptions) (0), 0, 0);
```

```
gtk_widget_set_size_request (button10,40,30);

button11 = gtk_button_new_with_mnemonic("log");          /*log*/
gtk_widget_show (button11);/*常显示，不变*/
gtk_table_attach (GTK_TABLE (table2), button11, 2, 3, 1, 2,
                    (GtkAttachOptions) (GTK_FILL),
                    (GtkAttachOptions) (0), 0, 0);

button12 = gtk_button_new_with_mnemonic("n!");           /*n!*/
gtk_widget_show (button12);/*常显示，不变*/
gtk_table_attach (GTK_TABLE (table2), button12, 2, 3, 2, 3,
                    (GtkAttachOptions) (GTK_FILL),
                    (GtkAttachOptions) (0), 0, 0);

button13 = gtk_button_new_with_mnemonic("1/x ");          /*1/x*/
gtk_widget_show (button13);/*常显示，不变*/
gtk_table_attach (GTK_TABLE (table2), button13, 2, 3, 3, 4,
                    (GtkAttachOptions) (GTK_FILL),
                    (GtkAttachOptions) (0), 0, 0);

button14 = gtk_button_new_with_label("7");          /*数字按钮 7*/
gtk_table_attach (GTK_TABLE (table2), button14, 3, 4, 0, 1,
                    (GtkAttachOptions) (GTK_FILL),
                    (GtkAttachOptions) (0), 0, 0);
gtk_widget_set_size_request (button14,40,30);

button15 = gtk_button_new_with_mnemonic("4");          /*数字按钮 4*/
gtk_table_attach (GTK_TABLE (table2), button15, 3, 4, 1, 2,
                    (GtkAttachOptions) (GTK_FILL),
                    (GtkAttachOptions) (0), 0, 0);

button16 = gtk_button_new_with_mnemonic("1");          /*数字按钮 1*/
gtk_widget_show (button16);/*常显示，不变*/
gtk_table_attach (GTK_TABLE (table2), button16, 3, 4, 2, 3,
                    (GtkAttachOptions) (GTK_FILL),
                    (GtkAttachOptions) (0), 0, 0);

button17 = gtk_button_new_with_mnemonic("0");          /*数字按钮 0*/
gtk_widget_show (button17);/*常显示，不变*/
gtk_table_attach (GTK_TABLE (table2), button17, 3, 4, 3, 4,
                    (GtkAttachOptions) (GTK_FILL),
                    (GtkAttachOptions) (0), 0, 0);

button18 = gtk_button_new_with_mnemonic("A");          /*数字按钮 A*/
gtk_table_attach (GTK_TABLE (table2), button18, 3, 4, 4, 5,
                    (GtkAttachOptions) (GTK_FILL),
                    (GtkAttachOptions) (0), 0, 0);

button19 = gtk_button_new_with_mnemonic("8");        /*数字按钮 8*/
gtk_table_attach (GTK_TABLE (table2), button19, 4, 5, 0, 1,
```

```
                                (GtkAttachOptions) (GTK_FILL),
                                (GtkAttachOptions) (0), 0, 0);
gtk_widget_set_size_request (button19,40,30);

button20 = gtk_button_new_with_mnemonic("5");          /*数字按钮 5*/
gtk_table_attach (GTK_TABLE (table2), button20, 4, 5, 1, 2,
                                (GtkAttachOptions) (GTK_FILL),
                                (GtkAttachOptions) (0), 0, 0);

button21 = gtk_button_new_with_mnemonic("2");          /*数字按钮 2*/
gtk_table_attach (GTK_TABLE (table2), button21, 4, 5, 2, 3,
                                (GtkAttachOptions) (GTK_FILL),
                                (GtkAttachOptions) (0), 0, 0);

button22 = gtk_button_new_with_mnemonic("+/-");        /*+/-*/
gtk_widget_show (button22);/*常显示，不变*/
gtk_table_attach (GTK_TABLE (table2), button22, 4, 5, 3, 4,
                                (GtkAttachOptions) (GTK_FILL),
                                (GtkAttachOptions) (0), 0, 0);

button23 = gtk_button_new_with_mnemonic("B");          /*数字按钮 B*/
gtk_table_attach (GTK_TABLE (table2), button23, 4, 5, 4, 5,
                                (GtkAttachOptions) (GTK_FILL),
                                (GtkAttachOptions) (0), 0, 0);

button24 = gtk_button_new_with_mnemonic("9");          /*数字按钮 9*/
gtk_table_attach (GTK_TABLE (table2), button24, 5, 6, 0, 1,
                                (GtkAttachOptions) (GTK_FILL),
                                (GtkAttachOptions) (0), 0, 0);
gtk_widget_set_size_request (button24,40,30);

button25 = gtk_button_new_with_mnemonic("6");          /*数字按钮 6*/
gtk_table_attach (GTK_TABLE (table2), button25, 5, 6, 1, 2,
                                (GtkAttachOptions) (GTK_FILL),
                                (GtkAttachOptions) (0), 0, 0);

button26 = gtk_button_new_with_mnemonic("3");          /*数字按钮 3*/
gtk_table_attach (GTK_TABLE (table2), button26, 5, 6, 2, 3,
                                (GtkAttachOptions) (GTK_FILL),
                                (GtkAttachOptions) (0), 0, 0);

button27 = gtk_button_new_with_mnemonic(".");          /*小数点*/
gtk_widget_show (button27);/*常显示，不变*/
gtk_table_attach (GTK_TABLE (table2), button27, 5, 6, 3, 4,
                                (GtkAttachOptions) (GTK_FILL),
                                (GtkAttachOptions) (0), 0, 0);

button28 = gtk_button_new_with_mnemonic("C");          /*数字按钮 C*/
gtk_table_attach(GTK_TABLE (table2), button28, 5, 6, 4, 5,
                               (GtkAttachOptions) (GTK_FILL),
```

```
                    (GtkAttachOptions) (0), 0, 0);

button29 = gtk_button_new_with_mnemonic("/");           /*除法*/
gtk_widget_show (button29);/*常显示，不变*/
gtk_table_attach (GTK_TABLE (table2), button29, 6, 7, 0, 1,
                    (GtkAttachOptions) (GTK_FILL),
                    (GtkAttachOptions) (0), 0, 0);
gtk_widget_set_size_request (button29,40,30);

button30 = gtk_button_new_with_mnemonic("*");           /*乘法*/
gtk_widget_show (button30);/*常显示，不变*/
gtk_table_attach (GTK_TABLE (table2), button30, 6, 7, 1, 2,
                    (GtkAttachOptions) (GTK_FILL),
                    (GtkAttachOptions) (0), 0, 0);

button31 = gtk_button_new_with_mnemonic("-");           /*减法*/
gtk_widget_show (button31);/*常显示，不变*/
gtk_table_attach (GTK_TABLE (table2), button31, 6, 7, 2, 3,
                    (GtkAttachOptions) (GTK_FILL),
                    (GtkAttachOptions) (0), 0, 0);

button32 = gtk_button_new_with_mnemonic("+");           /*加法*/
gtk_widget_show (button32);/*常显示，不变*/
gtk_table_attach (GTK_TABLE (table2), button32, 6, 7, 3, 4,
                    (GtkAttachOptions) (GTK_FILL),
                    (GtkAttachOptions) (0), 0, 0);

button33 = gtk_button_new_with_mnemonic("D");     /*D*/
gtk_table_attach (GTK_TABLE(table2), button33, 6, 7, 4, 5,
                    (GtkAttachOptions) (GTK_FILL),
                    (GtkAttachOptions) (0), 0, 0);

button34 = gtk_button_new_with_mnemonic("CR");     /*CR*/
gtk_widget_show (button34);/*常显示，不变*/
gtk_table_attach (GTK_TABLE (table2), button34, 7, 9, 0, 1,
                    (GtkAttachOptions) (GTK_FILL),
                    (GtkAttachOptions) (0), 0, 0);
gtk_widget_set_size_request (button34,80,30);

button35 = gtk_button_new_with_mnemonic("And");     /*And*/
gtk_widget_show (button35);/*常显示，不变*/
gtk_table_attach (GTK_TABLE (table2), button35, 7, 8, 1, 2,
                    (GtkAttachOptions) (GTK_FILL),
                    (GtkAttachOptions) (0), 0, 0);

button36 = gtk_button_new_with_mnemonic("Or");     /*Or*/
gtk_widget_show (button36);/*常显示，不变*/
gtk_table_attach (GTK_TABLE (table2), button36, 7, 8, 2, 3,
                    (GtkAttachOptions) (GTK_FILL),
                    (GtkAttachOptions) (0), 0, 0);
```

```
button37 = gtk_button_new_with_mnemonic("Mod");     /*Mod*/
gtk_widget_show (button37);/*常显示，不变*/
gtk_table_attach (GTK_TABLE (table2), button37, 7, 8, 3, 4,
                  (GtkAttachOptions) (GTK_FILL),
                  (GtkAttachOptions) (0), 0, 0);

button38 = gtk_button_new_with_mnemonic("E");     /*E*/
gtk_table_attach (GTK_TABLE(table2), button38, 7, 8, 4, 5,
                  (GtkAttachOptions) (GTK_FILL),
                  (GtkAttachOptions) (0), 0, 0);

button39 = gtk_button_new_with_mnemonic("Not");      /*Not*/
gtk_widget_show (button39);/*常显示，不变*/
gtk_table_attach (GTK_TABLE (table2), button39, 8, 9, 1, 2,
                  (GtkAttachOptions) (GTK_FILL),
                  (GtkAttachOptions) (0), 0, 0);

button40 = gtk_button_new_with_mnemonic("Xor");     /*Xor*/
gtk_widget_show (button40);/*常显示，不变*/
gtk_table_attach (GTK_TABLE (table2), button40, 8, 9, 2, 3,
                  (GtkAttachOptions) (GTK_FILL),
                  (GtkAttachOptions) (0), 0, 0);

button41 = gtk_button_new_with_mnemonic("Int");     /*Int*/
gtk_widget_show (button41);/*常显示，不变*/
gtk_table_attach (GTK_TABLE (table2), button41, 8, 9, 3, 4,
                  (GtkAttachOptions) (GTK_FILL),
                  (GtkAttachOptions) (0), 0, 0);

button42 = gtk_button_new_with_mnemonic("F");     /*F*/
gtk_table_attach (GTK_TABLE(table2), button42, 8, 9, 4, 5,
                  (GtkAttachOptions) (GTK_FILL),
                  (GtkAttachOptions) (0), 0, 0);

/*下面是创建 4 个单选按钮，并将“十进制”按钮设置为默认选中*/
radio = gtk_radio_button_new_with_label (NULL,
                        g_locale_to_utf8("十六进制",-1,NULL,NULL,NULL));
g_signal_connect(GTK_OBJECT(radio),"clicked",
                        G_CALLBACK(on_clicked),"Hex");
gtk_table_attach (GTK_TABLE (table1), radio, 0, 1, 1, 2,
                  (GtkAttachOptions) (GTK_EXPAND | GTK_FILL),
                  (GtkAttachOptions) (0),0,0);
gtk_widget_show (radio);
group = gtk_radio_button_group (GTK_RADIO_BUTTON(radio));
radio = gtk_radio_button_new_with_label (group,
                   g_locale_to_utf8("十进制",-1,NULL,NULL,NULL));
g_signal_connect(GTK_OBJECT(radio),"clicked",
                   G_CALLBACK(on_clicked),"Dec");
gtk_toggle_button_set_active(GTK_TOGGLE_BUTTON(radio),TRUE);
```

```
/*十进制 radio 设置为默认选中状态*/
gtk_table_attach (GTK_TABLE (table1), radio, 1, 2, 1, 2,
                    (GtkAttachOptions) (GTK_EXPAND | GTK_FILL),
                    (GtkAttachOptions) (0),0,0);
gtk_widget_show (radio);
group = gtk_radio_button_group (GTK_RADIO_BUTTON(radio));
radio = gtk_radio_button_new_with_label (group,
                    g_locale_to_utf8("八进制",-1,NULL,NULL,NULL));
g_signal_connect(GTK_OBJECT(radio),"clicked",
                    G_CALLBACK(on_clicked),"Oct");
gtk_table_attach (GTK_TABLE (table1), radio, 2, 3, 1, 2,
                    (GtkAttachOptions) (GTK_EXPAND | GTK_FILL),
                    (GtkAttachOptions) (0),0,0);
gtk_widget_show (radio);
group = gtk_radio_button_group (GTK_RADIO_BUTTON(radio));
radio = gtk_radio_button_new_with_label (group,
                    g_locale_to_utf8("二进制",-1,NULL,NULL,NULL));
g_signal_connect(GTK_OBJECT(radio),"clicked",
                    G_CALLBACK(on_clicked),"Bin");
gtk_table_attach (GTK_TABLE (table1), radio, 3, 4, 1, 2,
                    (GtkAttachOptions) (GTK_EXPAND | GTK_FILL),
                    (GtkAttachOptions) (0),0,0);
gtk_widget_show (radio);
addsignal ();          /*添加界面的信号与事件*/
gtk_widget_show (window);
gtk_main ();
return 0;
}
```

14.4 软件使用效果演示

在主函数中，我们已将所有的模块与函数以预处理命令的方式包含在 main.c 中了，所以使用下面的命令即可使用 gcc 编译器来编译整个计算器程序：

```
#gcc –o main main.c `pkg-config --libs --cflags gtk+-2.0`
```

运行可执行文件 main，便得到了计算器的初始界面，如图 14.1 所示。

```
#./main
```

下面简单向读者演示一下计算器的运行。首先单击“2”按钮，再单击“+/-”按钮，即输入的数值为-2，此时单击“log”按钮，在文本框中出现了“对数必须为正数”的提示信息，如图 14.2 所示。此时可以单击“CR”按钮来清除错误的输入。



图 14.1 计算器初始十进制界面



图 14.2 出错提示信息

选中“十六进制”单选按钮，计算器显示十六进制的界面，如图 14.3 所示。此时便可以进行十六进制数的计算了。

选中“八进制”单选按钮，计算器显示八进制的界面，如图 14.4 所示。此时便可以进行八进制数的计算了。



图 14.3 十六进制界面

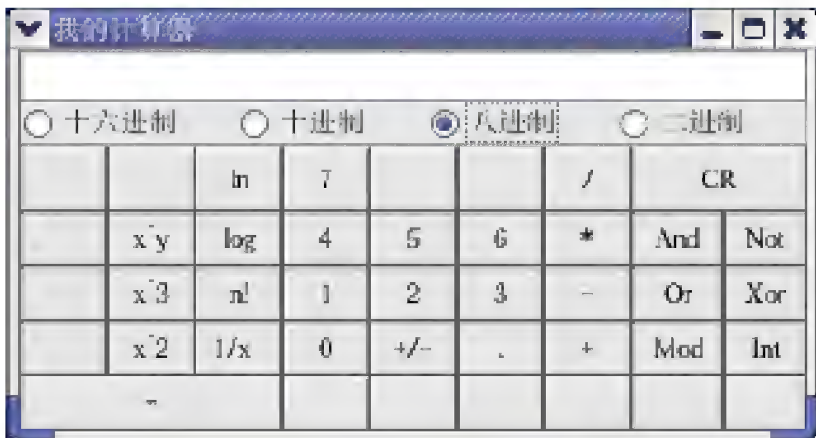


图 14.4 八进制界面

选中“二进制”单选按钮，计算器显示二进制的界面，如图 14.5 所示。此时便可以进行二进制数的计算了。

比如，单击按钮依次输入“1100101011”→“And”→“01110101”→“=”，可以得到以下的输出结果，如图 14.6 所示。



图 14.5 二进制界面



图 14.6 二进制数的运算

鉴于篇幅有限，关于计算的例子在此不一一列举，比如还有十六进制、八进制、二进制的小数运算(这在 Windows 下的计算器中是不支持的)，读者可亲自试验。本计算器运行稳定可靠，计算精确无误。希望读者能从本章的实例讲解中仔细体会 GTK+图形界面编程的方法，以及 GTK+信号与信号处理事件(在第 12 章中我们称为回调函数)的使用。

14.5 本章小结

计算器是每个计算机使用者都很熟悉的工具，但是如何设计自己的计算器，却不是一件很容易的事。本章带领读者设计了一个运行在 Linux 下的图形界面的计算器工具软件，通过掌握该软件的设计，读者应该能够进一步掌握 GTK+图形界面编程的方法与技巧，尤其是 GTK+中信号与回调函数的使用方法。

第15章

Linux平台下聊天软件的设计

网络的诞生从某种意义上来说改变了我们的生活，网络聊天软件也随之而来。熟悉Windows的用户不会对Windows下常用聊天软件感到陌生，比如QQ、MSN。此外在Internet上，还有ICQ、Gtalk、OICQ等网络聊天软件，最初的网络聊天软件只具有简单的文本界面，功能也十分有限，随着网络技术的不断发展，拥有更多功能和美观界面的聊天程序走进了我们的生活，越来越多的人将网络聊天软件作为日常生活交流和通信的工具。本章将带领读者设计一个Linux平台下的基于图形界面的网络聊天软件，该软件使用GTK+图形开发库和C程序语言，通信协议使用面向连接的TCP。



本章内容：

◎ 软件功能概述。

◎ Glade集成开发工具简介。

◎ 软件功能模块的划分。

◎ 服务器程序的具体实现。

◎ 客户端程序的具体实现。

15.1 软件功能概述

事实上，在本书的第 12 章“网络编程”中，已经给出了一个简单的基于 UDP 协议的文本通信程序(我们暂且不称它为聊天软件)，读者可回顾 12.4.2 小节。在 12.4.2 小节的实例中，客户端与服务器端可以互相发送/接收字符串。而本章将要向读者讲解的是基于 TCP 协议的图形化界面、功能较为齐全的聊天软件，具有一定的实用价值，甚至可以使用它来作为自己的聊天工具。

对软件功能需求的深入理解，是程序开发工作获得成功的前提条件，它对目标项目提出了完整、准确、清晰、具体的要求。本节现简要概述该聊天软件的客户端与服务器端的功能需求及特点。

15.1.1 服务器端功能需求

首先，服务器端应具有添加注册用户到数据库的功能。所有需要使用此聊天软件的用户都必须先在服务器端注册自己的用户名和密码。此外，服务器端也可以删除已注册的用户。

其次，当服务器端的聊天进程运行时，服务器需要能够同时连接很多个用户，并能提供给这些连接用户所需要的任务处理请求，这就要求服务器能同时处理多个 socket 连接。服务器同时处理多个客户机的连接请求及数据通信如图 15.1 所示，需要注意的是，这种模式仍然属于客户机/服务器(C/S)的连接模式。服务器负责向各个客户端发布系统消息，以及接受来自客户端的各种信息并分别处理。这些消息包括用户的聊天信息，以及用户的在线、离线状态信息等。因此，服务器端还必须具有保存和实时获取用户信息的功能。

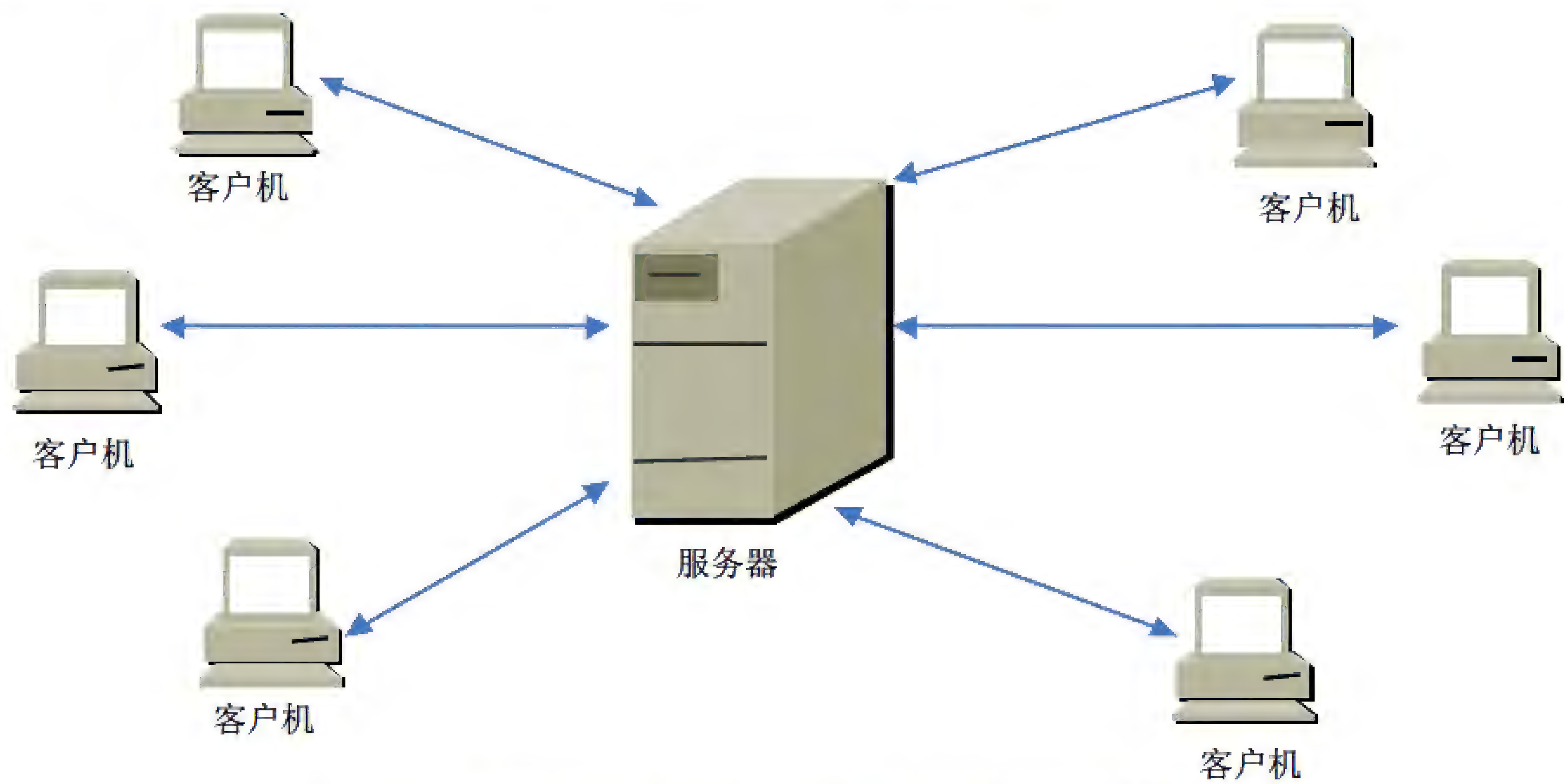


图 15.1　服务器同时处理多个客户机的连接请求及数据通信

服务器模型一般分为循环服务器和并发服务器，循环服务器一次只能处理一个连接，也就是说同一时间只能由一个用户连接到服务器进行消息处理，这种情况是不被允许的。

因此我们将采用多线程方式的并发服务器来设计服务器端，这样将能从很大程度上提高服

务器的运行效率。

另外，服务器端不需要复杂的图形界面。基于字符的方式可使服务器端的工作量大大减小。

15.1.2 客户端功能需求

客户端只需要连接到服务器便可以进行任务的处理工作，因此客户端的主要性能要求为图形界面运行的稳定性和对出错信息的及时反映。当一个窗体出现问题时能够及时地处理，并让主程序不受影响。

为了开发出符合要求的网络聊天软件，客户端应具有以下的功能。

1. 用户登录功能

用户登录功能实现对登录用户的身份验证。用户登录时需要填写的信息有3个：Server Ip(欲连接到的服务器 IP)、User Id(用户名)、Password(密码)。如果用户名或密码错误，则返回出错信息对话框。若验证正确则成功登录。

需要注意的是，在用户登录时，用户欲连接到的服务器必须是在线(on-line)的，事实上，在整个聊天软件运行的过程中，服务器必须是一直在线的。也就是说，服务器程序的套接口必须一直处于监听的状态(参照第 12 章)，等待客户机的连接请求。当与某一个客户机建立连接以后，服务器也必须一直处于接收与发送数据信息的状态。所以，用户登录时，若在“Server Ip”栏中填写一个并没有启动服务器进程的 IP 地址，同样无法登录，此时会自动弹出出错信息对话框，提示用户重新输入。

2. 添加新用户的功能

用户可以添加服务器端的数据库中已存在的任何用户，作为自己的聊天好友。单击聊天软件主界面的“Add”按钮，在弹出的“Add contacts”对话框中的文本框输入对方的用户名(User Id)，便可添加对方(类似于 QQ 中“添加对方为好友”)，若此用户是已在服务器的数据库中注册记录的用户，则会成功添加。若此用户尚未在服务器端的数据库注册或已经被删除，则不会添加成功，返回错误对话框。

3. 一对一聊天功能

服务器接收各个客户机的信息，也可以向每一个客户机发送数据。当添加聊天好友成功后，双击选择想要聊天的好友，便可以通过服务器的转发，实现一对一的聊天。

4. 显示好友状态信息功能

服务器可以实时获取注册用户的信息，比如他们的在线、离线信息，并以系统消息的方式分发给每一位在线的用户。在客户端，当客户机接收到这些信息后，检查和更新自己好友的状态，好友在线时的状态显示为绿色的“√”，离线时的状态显示为红色的“×”。

5. 易用、美观的图形界面

作为聊天软件，客户端应该具有易用、美观的图形界面，用户通过鼠标单击便可完成相应

的操作。在聊天对话框中应该包含一个用户输入文本框，一个发送按钮，和一个聊天记录的文本框。在聊天记录文本框中，分别以不同的颜色显示用户自身和好友，以区分聊天信息。

15.1.3 错误处理需求

所有的应用程序在运行过程中都会出现出错的情况，这种错误可能来自于程序本身的 BUG 也可能是用户操作的失误所造成的。当有错误发生时，我们应该有一个很好的机制来保障错误能够及时地被排除。

因此，当应用程序出现了错误的时候我们就需要程序能提供给我们出错的信息，这样用户就能够很快地找出具体的出错原因，以便寻找合理的途径去解决它。

15.2 Glade 集成开发工具简介

Glade 是一个功能强大的 GTK+图形界面产生器。也就是说，Glade 是一个界面化的程序设计工具，和 Windows 系统下的 VB、VC++类似，可以灵活使用各种功能设计出程序的界面。

Glade 是面向 GTK+/Gtkmm 的图形界面开发工具。它向用户提供可视化的界面设计环境，并以 XML 文件的格式进行保存。Glade 提供了一个类库，在程序运行的时候，它可以通过读取 XML 文件而生成相应的程序界面，从而达到了程序代码逻辑与用户界面的完全分离。另一方面，有另一个工具(glade--)可以用来把这些 XML 文件直接生成相应的 Gtk+/Gtkmm 代码。对于比较小型的项目，这种方法是比较适用的。

这种通过 XML 对界面进行描述的特性极大地增强了程序的灵活性，界面的设计与代码的编写可由相关的人员进行。也许这是以后桌面应用程序开发的一个方向。

需要注意的是 Glade 并不是一个程序开发平台。设计出界面以后，Glade 可以自动生成程序界面的部分代码，自动生成工程编译文件，但这时需要用其他的编译器来编译程序的事件代码。Glade 可以生成 C、C++、Ada95、Python、Perl 等语言的界面代码。

在 Red Hat Linux 9.0 或 Fedora 等 Linux 版本中都默认安装了 Glade。使用 Glade 之前需要启动它，选择“主菜单”→“编程”→“Glade Interface Designer”命令，便可以打开 Glade。Glade 有工作区、工具栏、属性编辑器、构建树、剪贴板等多个工作面板。启动 Glade 时会自动弹出工作界面、工具栏和属性编辑器 3 个面板。

Glade 的工作界面如图 15.2 所示。单击“视图”菜单下的各个选项便可以显示或隐藏相应的工作界面。

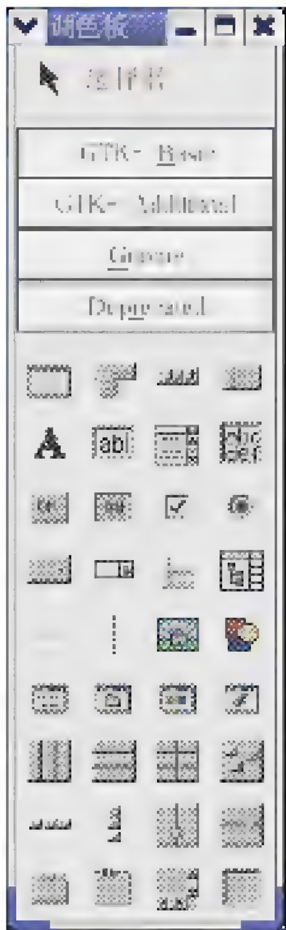
Glade 的工具栏也称为调色板，如图 15.3 所示。它提供了制作 GTK+图形界面的常用控件的调用接口，比如单击“窗口”工具(工具栏中的第一个小按钮)，Glade 便会在工作界面中新建一个窗口。

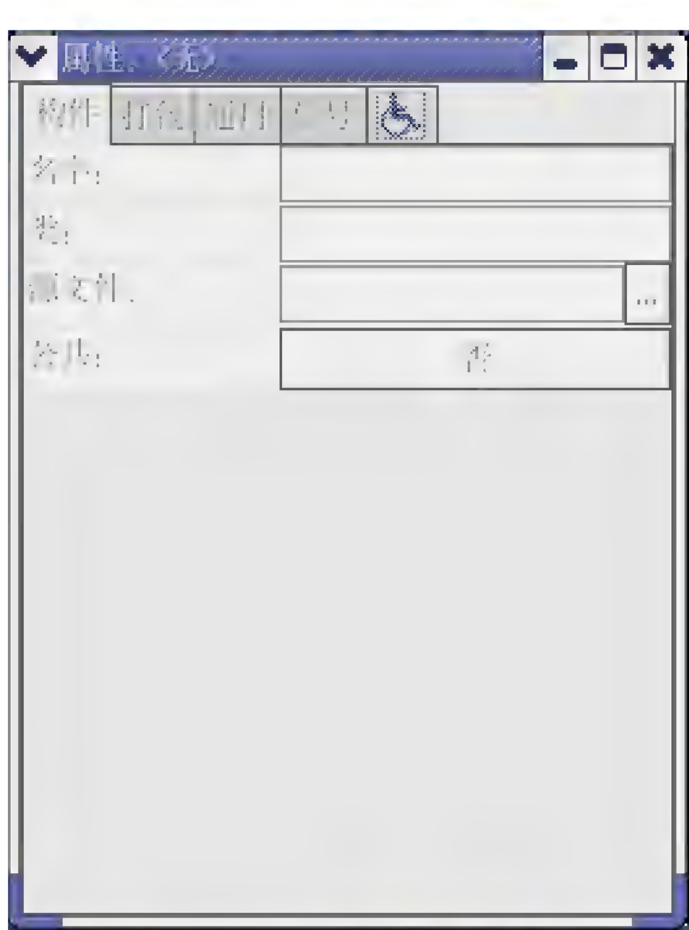
属性编辑器是 Glade 对程序中的各个控件进行属性设置的工具，其界面如图 15.4 所示。比如我们可以在这里设置刚才建立的那个窗口的标题、大小、初始位置等属性。对于按钮等控件，可以在“信号”选项下添加它的信号与事件响应。

本章所设计的聊天软件的图形界面正是使用 Glade 进行开发的，鉴于篇幅有限，在这里就不一一讲解 Glade 的使用方法，有兴趣的读者可参阅其他书籍，并通过实际操作来掌握。



图 15.2 Glade 的工作界面


图 15.3 Glade 的主工具栏


图 15.4 属性编辑器

15.3 软件功能模块划分

为实现网络聊天的功能，该软件采用 Socket 编程，基于 C/S 模式，服务器与客户端采用了 TCP/IP 方式连接，在设计聊天方案时，实行将所有信息发往服务器端，再由服务器进行分别处理的思路，服务器端是所有信息的中心。

15.3.1 服务器功能模块划分

在聊天软件运行的过程中，服务器的主要功能是负责向各个客户端发布系统消息和接受来自各个客户端的各种信息并分别处理。针对这些操作，服务器做了如下的模块划分，如图 15.5 所示。

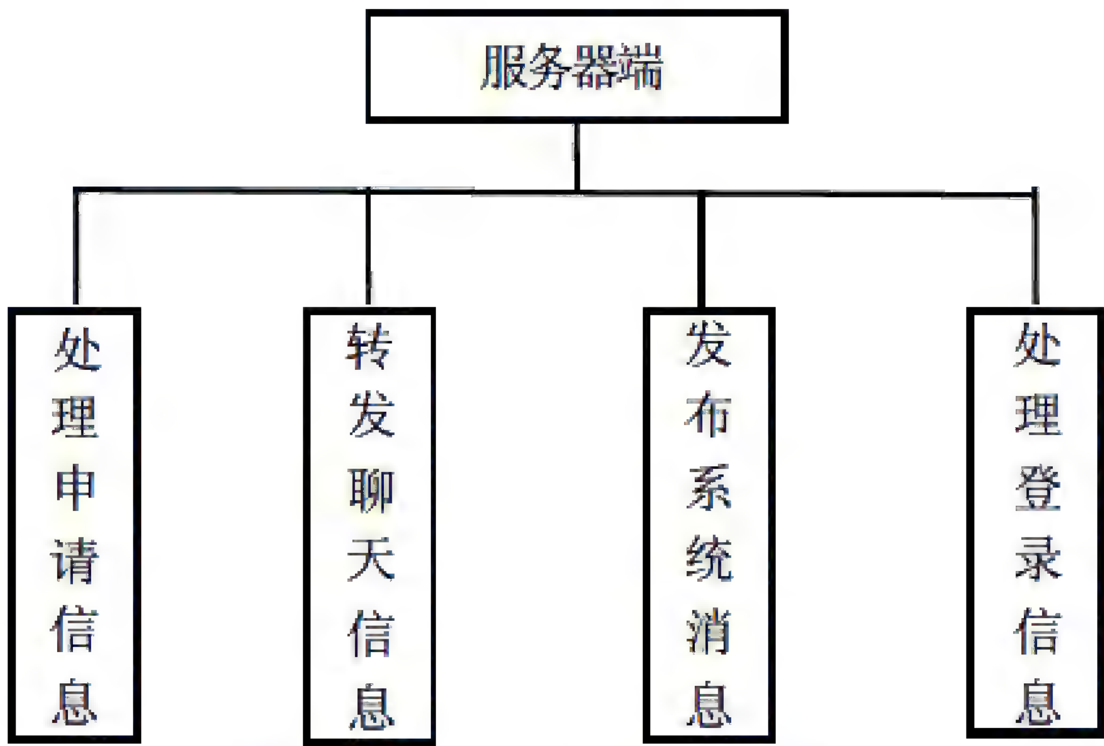


图 15.5 服务器端功能模型

➢ 处理登录信息模块：检查用户登录信息是否正确，并向客户端返回登录信息的验证情况。如果登录信息正确，就将当前在线的用户发送给该用户，并将该用户的状态发送给各在线用户，同时在服务器端显示出来。
➢ 转发聊天信息模块：转发消息给指定的用户。
➢ 处理申请信息模块：通过用户申请模块进行新用户的注册，保存该用户信息。
➢ 发布系统消息模块：将用户上下线的消息发给各客户端，并改写用户在服务器端和客户端的状态。

15.3.2 客户端功能模块划分

客户端主要负责处理用户的操作信息，当用户做出相应的动作时客户端应该能够及时地做出响应，当 GTK+图形程序检测到鼠标单击时，将触发一个事件，对该事件进行动作的定义和函数的编写便可完成相应的动作。因此，针对这些操作特性，将客户端模块进行如下划分，如图 15.6 所示。

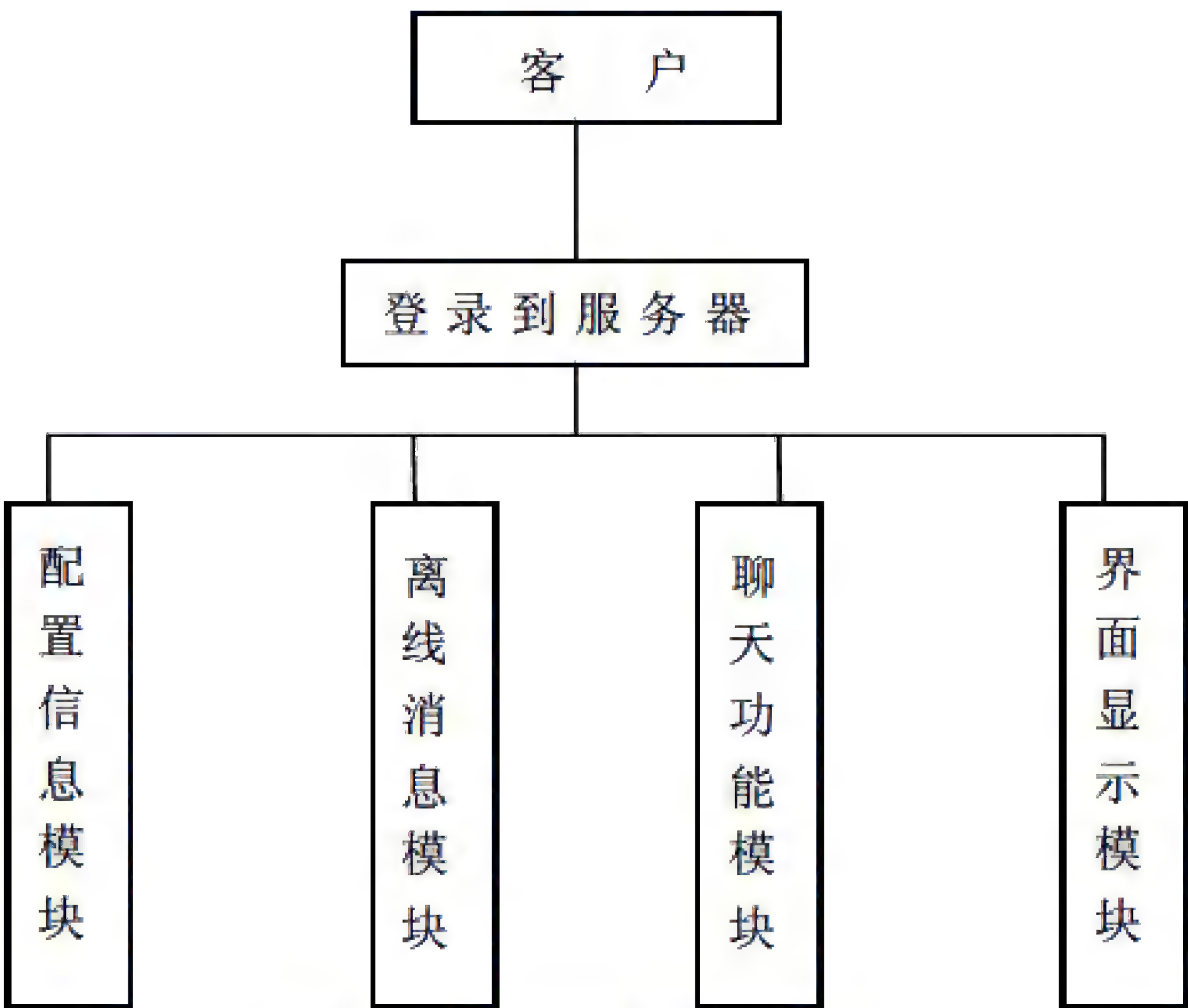


图 15.6 客户端功能模型

➢ 登录功能模块：建立与服务器的连接并登录，若登录时输入用户信息错误，则会显示登录出错对话框，提醒用户重新输入。
➢ 界面显示模块：将在线好友显示在好友列表中，并显示其当前状态。
➢ 聊天功能模块：处理各种聊天信息，并判断消息类型。
➢ 离线消息模块：接收和发送离线消息，对离线消息进行处理。
➢ 配置信息模块：提供基本的软件配置操作。

15.3.3 消息标识的定义

本章所设计的聊天软件的消息类型有很多种，在头文件中将它们定义成不同的常量，以便在不同的函数中调用它们。消息类型标识的定义如下：

```
#define GIF_LOGIN_MSG 1        /*用户登录*/
#define GIF_CALL_MSG 2         /*用户呼叫*/
#define GIF_ADDRLIST_MSG 3         /*更新联系人列表信息*/
#define GIF_ADD_CONTACTS_MSG 4         /*添加连接信息*/
```

```
#define GIF_DELETE_CONTACTS_MSG 5 /*删除连接信息*/
#define GIF_CHAT_MSG 6        /*聊天信息*/
#define GIF_DISCONNECT_MSG 7              /*用户下线消息*/
#define GIF_SUCCESS_N_ERROR_MSG 8   /*消息请求成功或失败信息*/
#define GIF_OFFLINE_REQUEST_MSG 9   /*发送离线消息*/
#define GIF_OFFLINE_MSG 10                /*离线消息*/
#define GIF_OFFLINE_DELETE_MSG 11   /*删除离线消息*/
```

用户发送消息成功或失败的定义如下：

```
#define GIF_ERROR_LOGIN_INCORRECT 101   /*错误的登录信息*/
#define GIF_SUCCESS_ADD_CONTACTS 102     /*添加连接信息成功*/
#define GIF_ERROR_ADD_CONTACTS 103      /*添加连接信息错误*/
#define GIF_SUCCESS_DELETE_CONTACTS 104    /*删除连接信息成功*/
#define GIF_ERROR_DELETE_CONTACTS_NOT_A_CONTACT 105    /*用户离线*/
#define GIF_ERROR_DELETE_CONTACTS_NOT_A_MEMBER 106      /*无指定用户*/
```

15.3.4 消息结构体的设计

不同的消息类型具有不同的数据结构，服务器或客户端也是通过这些不同的数据结构来区分不同的消息类型。下面分别给出该聊天软件中用到的各种不同类型消息的结构定义。

消息头的定义如下：

```
typedef struct _gifhdr_t
{
        unsigned int type;     /*消息类型*/
        unsigned int length;  /*消息的长度，以字节为单位*/
        char sender[10];       /*发送方*/
        char receiver[10];     /*接收方*/
unsigned int reserved;
}gifhdr_t;
```

用户信息结构的定义如下：

```
typedef struct _users_t
{
        char loginid[20];       /*用户名*/
        char password[20];  /*密码*/
}users_t;
```

在线用户信息结构的定义如下：

```
typedef struct _online_users_t
{
        char loginid[20];       /*用户名*/
        int sockfd;                /*在线用户客户机的套接字*/
}online_users_t;
```

用户联系人信息结构的定义如下：

```
typedef struct _user_contacts_t
{
        char loginid[20];
}user_contacts_t;
```

用户当前状态的定义如下：

```
typedef struct _user_status_t
{
        char loginid[20];
        unsigned int status;
}user_status_t;
```

离线消息存储时离线消息结构的定义如下：

```
typedef struct _offline_msgs_t
{
        char sender[20];
        char dateserial[20];
        unsigned int new;
        char message[1024];
}offline_msgs_t;
```

用户发送离线消息结构的定义如下：

```
typedef struct _offline_msgs_send_t
{
        char sender[20];
        char dateserial[20];
        unsigned int new;
        unsigned int length;
}offline_msgs_send_t;
```

15.4 服务器程序的具体实现

本节主要讲述服务器端程序的整体运作流程(整个软件的全部源代码在本书附带的光盘中给出)。通过本节的介绍，读者将能够对该聊天软件的服务器端的功能，以及其大致的工作流程有一个总体的认识。

15.4.1 服务器消息处理流程

服务器运行后即处于监听状态，当监听到有连接请求时服务器进入消息处理流程，因为服务器为并发服务器，所以可以同时对多个请求做出响应。其具体工作流程如图 15.7 所示。

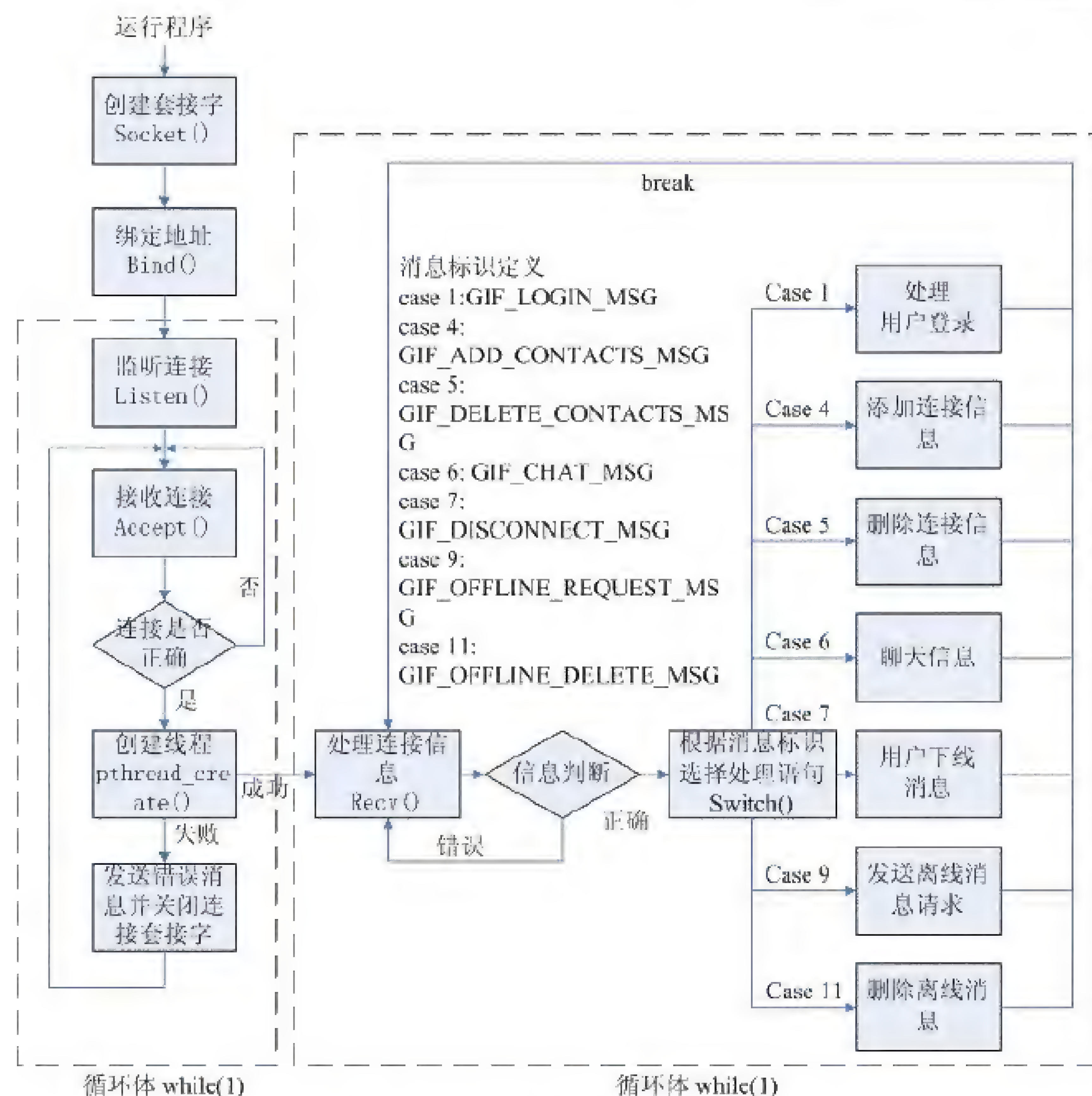


图 15.7　服务器工作流程图

15.4.2　服务器主要函数和变量

服务器的主体函数只负责了套接字的创建、绑定和监听任务，一旦服务器接收到了来自客户端的消息后，就会创建一个线程来处理该连接，当线程创建后它就会调用相应的函数对客户消息进行处理。

服务器端的主要变量包括：

- 套接口描述字：int sockfd, client_sockfd。
- 套接口地址结构：struct sockaddr_in servAddr, cliAddr。
- 线程 id 标识：pthread_t pthd。

服务器端的主要函数包括以下几个。

(1) 线程调用函数，它的定义方法如下：

```
void gif_handle_client (int client_sockfd);
```

该函数中有 7 个主要消息处理模块，这些模块针对不同的客户消息进行处理，将它们组合在一起便形成了一套完整的消息处理机制。在 15.4.1 小节中我们叙述了服务器的消息处理流程，而最后的消息处理部分就是由这 7 个模块来完成的。

(2) 发送联系人状态函数，函数定义方法如下：

```
void gif_send_clients_contact_list (chat *client_loginid,int client_sockfd, int type);
```

该函数的主要作用是当用户的在线状态标志发生变动时，服务器要及时地做出判断，并向该用户的联系人(聊天好友)发送该用户状态变更后的信息，以便对方能够及时地了解到联系人

的当前状态信息。

(3) 获取系统时间函数，函数定义方法如下：

```
char *gif_get_system_time ();
```

该函数取出当前系统时间并随消息一同发出，以便让用户能更好地了解到消息是何时发出的。

15.4.3 服务器消息处理模块的设计与实现

服务器的消息处理模块主要由 7 部分组成，下面分别对其进行论述。源代码可参见本书附带光盘中的程序 15.1：服务器消息处理源代码 gchat_server.c。

1. 用户登录消息处理模块

当服务器接收到用户消息并判断是登录消息后，服务器将根据用户所发送过来的用户名和密码到 users.db 数据库表中进行匹配，如果匹配成功则把该用户加入到 online.db 数据库表中，设置用户在线状态为真的同时调用发送联系人在线列表函数，向添加该用户为好友的用户发送该用户的上线信息。如果匹配不成功，服务器将发送登录失败消息给客户端，要求用户重新登录。

2. 添加好友信息处理模块

当服务器收到用户请求加好友的消息时，服务器首先在 users.db 中查找要被添加的用户是否存在，如果该用户存在则把该用户的信息存储到当前用户的好友文件中，同时在被添加好友的用户的_as.db 表中保存当前用户的信息。

3. 删除好友信息处理模块

当服务器接收到用户的该消息请求时，服务器根据用户所提供的好友用户名来删除被指定的好友，同时将用户联系人表进行更新，在被删除的好友的被添加好友表_as.db 中将当前用户删除。

4. 用户聊天信息转发的处理模块

当服务器判断用户所请求的信息为聊天消息后，服务器会根据用户所指定的聊天对象到 online.db 表中查找被指定用户是否在线，如果被指定用户在线则提取他的套接字地址结构，并按照该套接字信息将消息转发。如果用户不在线则将消息标识设定为离线消息，并将该消息写入被指定用户的离线消息文件中。

5. 用户下线消息处理模块

当服务器判断用户所请求的信息为下线消息后，服务器将把用户从 online.db 中删除，设置该用户状态为下线，同时调用发送联系人状态函数将标志位设置为离线，并向该用户的联系人发送。

6. 离线消息处理模块

当服务器判断用户所请求的信息为读取离线消息后，服务器会去读用户的离线消息文件，如果该文件中有被标识为新的离线消息时，服务器将把该消息提取出来发送给用户，同时把该消息标识为已读。

7. 删除离线消息处理模块

如果客户做出了删除离线消息的动作，那么服务器端将会根据用户的发送信息把该用户的

离线消息文件中的离线消息删除。

15.4.4 服务器数据存储的方法

该服务器采用文件作为数据存储的对象。之所以采用这种存储方式是因为该服务器作为一个小型的聊天软件的服务器，本身所要求保存的数据量并不大，且用户数量也是比较有限的，在这种情况下采用文件方式对信息进行读取在速度和开销上都要有一些优势，且便于管理员管理用户。

在编程时主要使用 fopen、fread、fwrite、fclose 等函数对文件进行操作，被打开文件的操作权限由 fopen 的参数来决定，读取控制由 fread 来完成，fwrite 主要负责向文件中写入新的信息。

服务器上主要存放 5 类用户信息文件：

- 用户信息数据文件：users.db。该文件中保存了用户的 loginid 和 password。
- 在线用户数据文件：online.db。该文件中保存了当前在线用户的 loginid 和 socket 信息。
- 用户拥有好友文件：loginid.db。该文件中保存了当前用户所添加的好友信息。
- 被添加为好友文件：loginid_as.db。该文件中保存了有哪些用户把当前用户添加为好友。
- 离线消息存储文件：loginid_off.db。该文件中保存了其他用户发给当前用户的离线消息。

15.4.5 用户注册流程

管理员在服务器一端统一注册用户，然后将账号分发给各个用户，申请用户为单独程序完成。当新用户注册好后，系统会自动创建 3 个用户信息表，用来保存相应的客户信息。用户注册流程如图 15.8 所示。程序源代码参见本书附带光盘中的程序 15.2：服务器注册、删除用户源代码 addUsers.c。

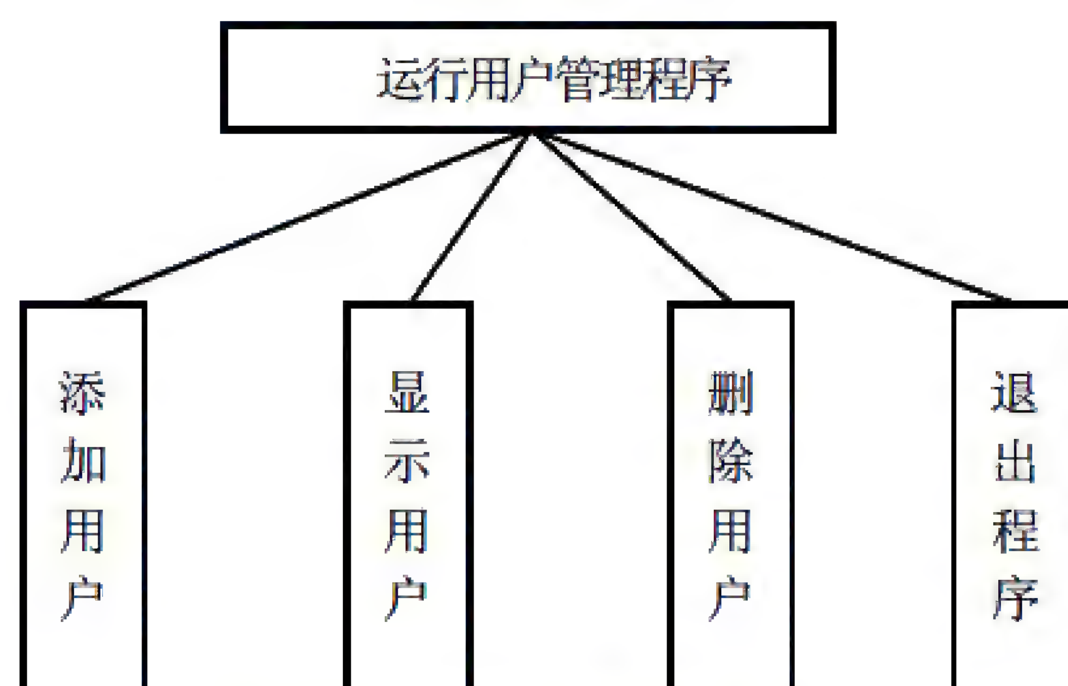


图 15.8 用户注册流程图

15.5 客户端程序的具体实现

本节主要讲述客户端程序的整体运作流程。客户端最主要的任务是建立图形化的用户操作接口，对用户输入的信息、服务器端发送过来的数据及时地做出响应，以及向服务器发送信息等。

15.5.1　客户端操作流程

这款聊天软件的客户端采用图形化用户接口方式，GTK+图形界面的主要特点就是事件触发，当处理引擎接收到某一窗体消息时，处理引擎就会按照事先编写好的函数做相应的处理动作，图 15.9 所示总体上描述了客户端软件的操作处理流程。

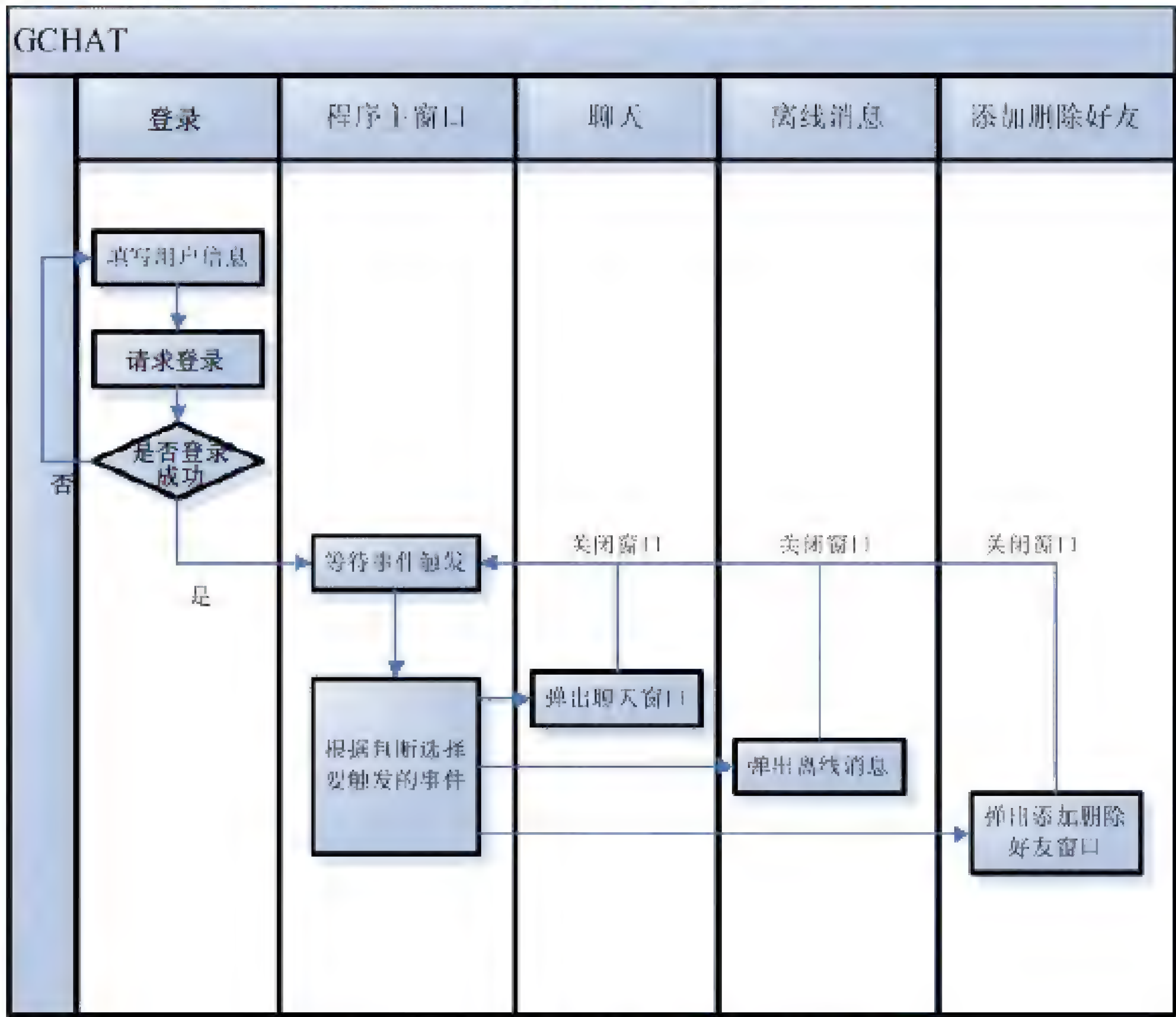


图 15.9　客户端操作处理流程图

15.5.2　客户端发送和接收消息流程

客户端运行后首先要用户登录到服务器，如果登录成功，服务器会将在线的用户联系人列表发送给客户端，此时客户端即可显示有哪些好友在线而哪些是离线的。当用户单击在线好友时便会弹出一个聊天信息窗体，该窗体用于发送和接收用户的聊天信息。当用户单击离线用户时便会触发一个离线消息事件，该事件通过服务器转发方式发送给离线用户，当对方上线时便可查看该离线消息。具体流程如图 15.10 所示。

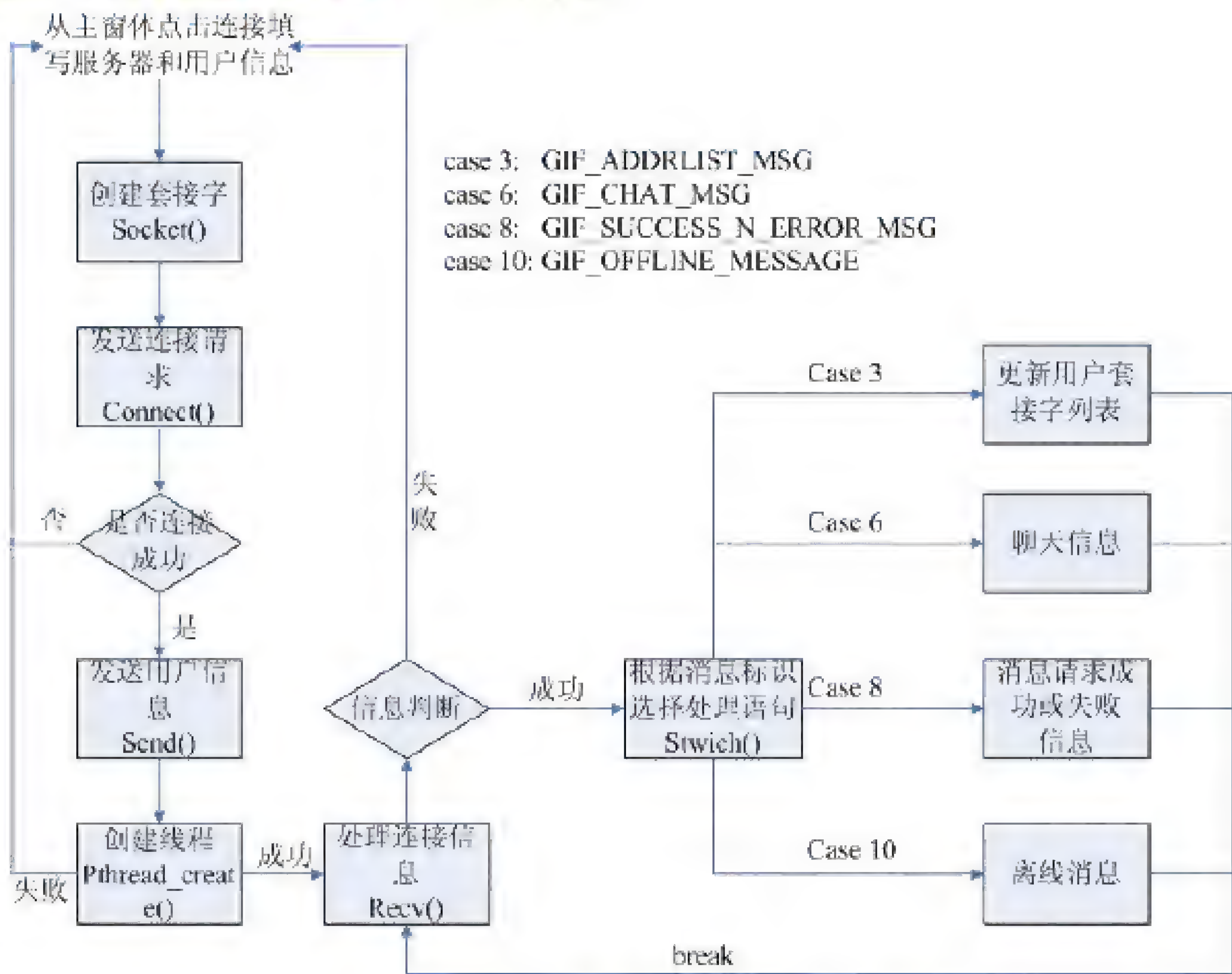


图 15.10　客户端发送/接收消息流程

15.5.3 客户端主要函数和变量

客户端需要提供给用户简易方便的图形界面操作接口，它的主要变量定义如下：

```
GtkWidget *tree;            /*用于显示客户端联系人列表的树视图*/
GtkWidget *offline_tree;    /*用于显示离线消息的树视图*/
GtkWidget *chat_window[100];    /*用于保存聊天窗体的编号*/
GtkListStore *offline_temp_store    /*定义一个离线消息数据管理类型*/
GtkTreeIter offline_selected_iter;    /*定义一个查询离线消息的迭代器*/
GtkTreeModel *offline_selected_model;    /*定义离线消息查询数据管理接口*/
GtkCellRenderer *renderer;        /*定义一个数据表现形式绘制类型*/
```

另外还有两个枚举类型的变量。下面的枚举类型定义了用户联系人视窗中的列标识：

```
enum
{
    CONTACTS_COLUMN_TEXT,
    STATUS_COLUMN_PIXMAP
};
```

下面的枚举类型定义了用户离线消息查看窗口中的列标识：

```
enum
{
    OFFLINE_NEW_PIXMAP,
    OFFLINE_SENDER_TEXT,
    OFFLINE_DATESERIAL_TEXT,
    OFFLINE_MSG_TEXT,
    OFFLINE_COLUMNS
};
```

下面的函数用于设置指定控件的敏感度，定义方法如下：

```
gtk_widget_set_sensitive (widget, TRUE);
```

参数 widget 表示一个指定的控件，参数 TRUE 表示可以单击，即单击时将引起相应的响应事件发生，FALSE 则表示不能单击。

gif_receive_messages()函数用于处理从服务器端发送过来的消息，根据信息包所携带的消息类型信息选择消息处理模块。定义方法如下：

```
void gif_receive_messages (int server_sockfd);
```

当用户单击联系人时会产生一个消息事件，程序根据用户所选择的联系人生成聊天窗口，并对该窗体进行初始化。该初始化函数的定义方法如下：

```
void gif_call_client_for_chat (GtkTreeSelection *selection, gpointer data);
```

另外，在系统调用线程时为保证线程安全，需要前后分别调用 gdk_threads_enter ()和 gdk_threads_leave()函数。例如：

```
gdk_threads_enter ();
```

```
    gtk_widget_show (create_msgbox ("error", "Server Disconnected"));
    gdk_threads_leave ();
```

15.5.4 客户端功能模块的设计与实现

下面介绍客户端功能实现的主要函数调用，鉴于篇幅有限，附录中只给出了客户端主函数的源代码，其他各个模块读者可参见本书光盘中的内容。

1. 用户认证模块

当用户启动程序后首先需要用户登录到服务器才能进一步的对软件进行操作，这一部分即为用户认证模块。主要用到的函数有 connect()、send()和 pthread_create()调用。

用户认证模块在登录窗口中收集服务器的 IP 地址、用户的用户名和密码。当用户单击登录后，系统将用户输入的 IP 地址写入 Socket 地址结构，然后对套接口地址结构进行绑定。同时将用户名写入消息头结构体，并将密码随同数据字段一起发送给服务器。如果登录成功，则创建线程来维护和服务器的连接。之后，程序进入主体运行状态，各功能模块被激活。

2. 界面显示模块

当用户成功登录后，在联系人列表中将会显示用户的联系人信息，在离线消息窗口中会显示接收到的离线消息，这两个视图均是用列表视图来实现的。

用户列表和离线消息列表的构建是通过下面的函数来实现的：

```
    store = gtk_tree_store_new (2, G_TYPE_STRING, GDK_TYPE_PIXBUF);
    /*新建一个数据的存储模型*/
    tree = gtk_tree_view_new_with_model (GTK_TREE_MODEL (store)); /*新建一个管理视图*/
    gtk_container_add (GTK_CONTAINER (scrolledwindow1), tree);
    /*将视图添加到指定的容器中*/
```

通过以上的 3 个步骤，就会在容器中生成一个可供用户查看的滚动视图。

但这个视图中并没有定义列信息，也就是说，这是一个空的无任何用处的列表视图，要将用户的信息显示出来，还需要使用以下的函数：

```
    renderer = gtk_cell_renderer_pixbuf_new ();          /*新建一个图像类型的绘制方式*/
```

下面的函数作用是给列添加新的属性，参数 STATUS_COLUMN_PIXMAP 指定了行信息，“pixbuf”指定了绘制类型为图像，renderer 指定了绘制方式。函数定义如下：

```
    column = gtk_tree_view_column_new_with_attributes ("Status", renderer, "pixbuf",
STATUS_COLUMN_PIXMAP, NULL);
```

将建好的列添加到树视图中：

```
    gtk_tree_view_append_column (GTK_TREE_VIEW (tree), column);
```

通过以上 3 个步骤一个视图中就有了列，进行相同的操作可创建多个列值，用户联系人视图中只建了两个列分别显示联系人姓名和联系人状态。

以上所创建的视图和存储空间都还是空的，要想将数据从其中显示出来还需要使用另外两个函数。下面的函数负责从指定的存储模型中取出新行的 iter：

```
gtk_tree_store_append (store, &parent_iter, NULL);
```

设置新添加行的值：

```
gtk_tree_store_set (store, &parent_iter,CONTACTS_COLUMN_TEXT, "Available", -1);
```

iterator 是系统提供的一种访问一个容器(container)对象中各个元素，而又不需暴露该对象的内部细节的方法。

通过以上两个函数，便可以在视图中创建新的显示行。

3. 消息处理模块

当客户端和服务器建立连接后，客户端会创建一个专门的线程来维护客户端和服务器之间的连接，该线程调用函数根据从服务器接收到的消息类型来选择消息处理模块。该模块主要处理以下几种消息类型：

(1) 更新联系人列表信息。当判断接收到的消息为 GIF_ADDRLIST_MSG 时，程序将对联系人视图进行刷新以便及时地显示联系人的当前状态。实现刷新的方法是，先将以前的树移除然后按照新的联系人状态进行树的绘制，具体的编写方法在前面的界面显示模块中已给出，这里就不再重复了。

(2) 聊天信息。当接收服务器发送的消息的类型为 GIF_CHAT_MSG 时，程序首先要确定是谁发出的聊天请求，然后判断聊天窗口是否开启，如果未开启则将开启标识设为真，然后打开聊天窗口。同时设置用户和联系人的显示颜色，以便区分聊天消息的来源。

(3) 消息请求成功或失败信息。如果接收到了 GIF_SUCCESS_N_ERROR_MSG 类型的消息，则程序还会进一步的根据另外一个标识符判断是何种系统消息，并通过系统消息窗口显示给用户。

(4) 离线消息。如果接收到的消息类型为 GIF_OFFLINE_MSG，程序会按照界面显示模块中对树视图的创建步骤那样，创建一个显示离线消息的视图用来显示离线消息。

以上的几大模块构成了客户端功能处理的主体，大部分的操作和消息处理都是由这些模块来完成的。

15.6 聊天软件使用效果演示

本章所设计的聊天软件的源代码以压缩包的形式存在本书附带的光盘中，对压缩包进行解压以后，在当前解压目录 gchat 下会产生两个子目录 gchat_client 和 gchat_server。

首先打开一个 shell 终端，进入 gchat_server 子目录，使用 gcc 编译服务器端的添加用户程序 addUsers.c，并生产可执行文件 addUsers：

```
[root@localhost gchat_server]# gcc -o addUsers addUsers.c
```

运行程序，将会出现 4 个选择项：添加、显示、删除和退出。可以通过输入不同的数字来选择不同的功能。如下所示，这里我们依次注册了 Jame、Lucy、Lily、Lilei 4 个用户，他们的密码均设置为 123456。

```
[root@localhost gchat_server]# ./addUsers
Select an option :
            1 . Add
            2 . Display
            3 . Delete
            4 . Exit
Enter ur choice : 1                        #选择 1 添加注册新用户
Enter the name to be added : Jame          #用户名 Jame
Enter the password : 123456                #密码
Select an option :
            1 . Add
            2 . Display
            3 . Delete
            4 . Exit
Enter ur choice : 1                        #选择 1 添加注册新用户
Enter the name to be added : Lucy          #用户名 Lucy
Enter the password : 123456                #密码
Select an option :
            1 . Add
            2 . Display
            3 . Delete
            4 . Exit
Enter ur choice : 1                        #选择 1 添加注册新用户
Enter the name to be added : Lily          #用户名 Lily
Enter the password : 123456                #密码
Select an option :
            1 . Add
            2 . Display
            3 . Delete
            4 . Exit
Enter ur choice : 1                        #选择 1 添加注册新用户
Enter the name to be added : Lilei         #用户名 Lilei
Enter the password : 123456                #密码
Select an option :
            1 . Add
            2 . Display
            3 . Delete
            4 . Exit
Enter ur choice : 2                        #选择 2 显示当前已注册的用户

Available Names :
Jame
Lucy
Lily
Lilei
Select an option :
            1 . Add
            2 . Display
            3 . Delete
            4 . Exit
```

```
Enter ur choice : 4                    #选择 4 退出
[root@localhost gchat_server]#
```

同样在 gchat_server 目录下，使用 gcc 编译服务器端的通信连接程序 gchat_server.c，并生成可执行文件 gchatserver：

```
[root@localhost gchat_server]# gcc -o gchatserver gchat_server.c
```

运行程序，得到以下输出：

```
[root@localhost gchat_server]# ./gchatserver
running gchat........
```

可以看到，服务器一直处于监听的状态，等待某一客户机的连接请求。

此时，打开第二个 Shell 终端，进入 gchat_client 目录下的 src 子目录，该目录下存放了客户端程序的源代码及相应的可执行文件。执行该目录下的可执行文件 gchat，便启动了客户端的聊天程序，命令如下：

```
[root@localhost zhangfan]# cd /gchat/gchat_client/src
[root@localhost src]# ./gchat
```

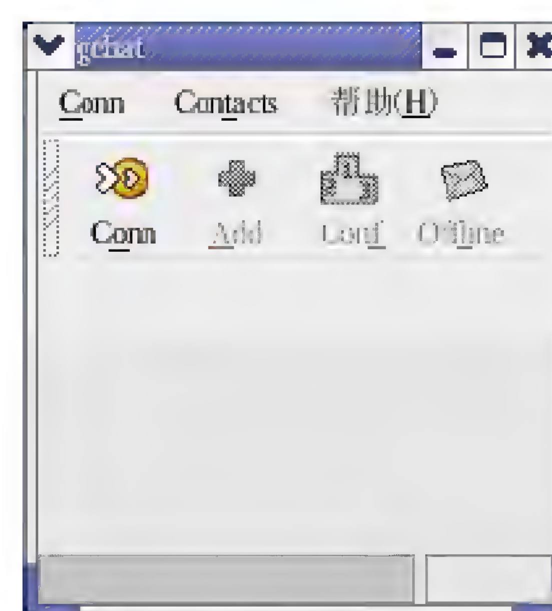


图 15.11　客户端程序主窗口

此时便出现了客户端程序的主界面窗口，如图 15.11 所示。

该窗体中包含了以下按钮：

- Conn：登录按钮，单击可激活登录窗口。
- Add：添加好友按钮。
- Conf：程序配置按钮。
- Offline：查看离线消息按钮。
- Conn 菜单：包含 Conn、Disconnect 和 Quit 按钮。
- Contacts 菜单：包含 Add、Delete 和 Offline 按钮。
- Help(帮助)菜单：包含 About 按钮。

界面的下方是联系人(好友)列表(稍后将会看到)：

- Status 列：显示联系人状态信息。
- Contacts 列：显示好友列表。Available 显示表示用户有好友，当用户无好友时显示 Not Available。

单击程序主窗口中的“Conn”按钮进行用户登录连接，用户登录界面如图 15.12 所示。



图 15.12　用户登录界面

由于是在同一台 Linux 主机下进行试验，这里的“Server Ip”填写为“localhost”，表示本机 IP，然后输入用户名和密码(在服务器端已注册)。单击“确定”按钮，或按“Enter”键，用户 Jame 便会成功登录。

此时查看第一个 shell(即服务器端)下多了一行输出，如下所示：

```
running gchat........
Jame - Login Correct
```

这表明用户 Jame 已经成功登录到服务器。

此时程序的主界面如图 15.13 所示。可以看到，联系人(好友)列表显示为“Not Available”，表明当前用户 Jame 还没有添加其他用户为自己的聊天好友。

单击图 15.13 中的“Add”按钮，添加已在服务器端成功注册的用户(比如 Lucy、Lily、Lilei 中的任何一个)为自己的好友，如图 15.14 所示。这里我们依次将 Lucy、Lily 和 Lilei 添加为 Jame 的好友。

单击“确定”按钮后会返回成功添加信息对话框，说明已成功添加对方为自己的聊天好友。若出错，则会弹出提示错误信息的对话框。添加成功的信息对话框如图 15.15 所示。



图 15.13　好友列表为空



图 15.14　添加好友对话框



图 15.15　成功添加好友

此时打开第三个 shell 终端，同样进入 gchat_client 目录下的 src 子目录，并运行可执行文件 gchat，按照上面的步骤使用用户 Lucy 登录到服务器，此时可以观察第一个 shell 终端下的输出为：

```
running gchat........
Jame - Login Correct
Lucy - Login Correct
```

说明用户 Lucy 也已经成功登录到服务器。

接下来是为用户 Lucy 添加自己的聊天好友，这里同样将其他 3 个用户都添加为 Lucy 的好友(同前面讲述的步骤一致)。添加成功后，可以看到两个用户的主程序界面分别如图 15.16 和图 15.17 所示。

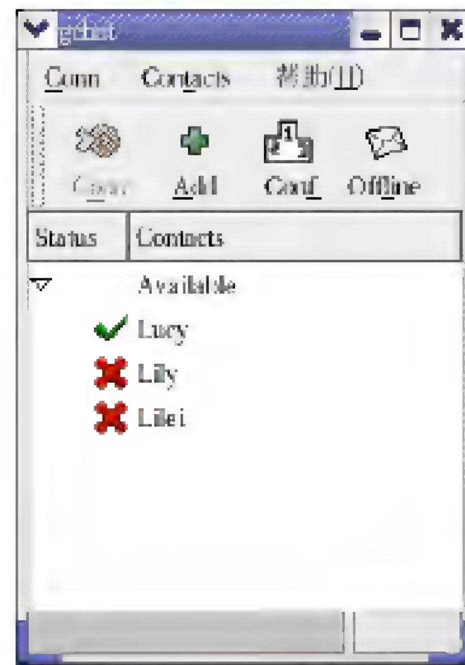


图 15.16　用户 Jame 的主界面

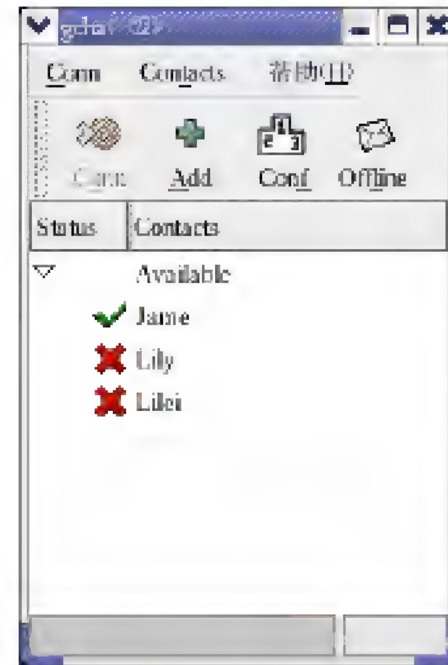


图 15.17　用户 Lucy 的主界面

当然，还可以添加更多的用户，只要这个用户是在服务器的数据库中成功注册的。从用户客户端的主界面中可以看到，在线好友(比如 Jame 的在线好友 Lucy)的状态显示为绿色的“√”，离线好友的状态显示为红色的“×”。

当使用鼠标单击在线好友的用户名时，便可以进行在线聊天了，例如用户 Jame 的聊天对话框如图 15.18 所示。红色显示的是用户自身，蓝色为联系人。在文本输入框输入聊天信息，单击“Send”按钮，或按“Enter”键即可发送给对方，聊天记录显示在上面的文本框中。



图 15.18　聊天对话框

鉴于篇幅有限，本聊天软件的其他功能就不在此一一列举，比如给离线好友发送消息等等，有兴趣的读者可以将本书附带的光盘中的源代码复制到自己的 Linux 主机中进行试验。

15.7 本章小结

通过该软件的设计，读者应该学到小型项目工程软件的模块划分方法，以及 Linux 下的 C 程序开发的步骤。在设计 Linux 下的纯软件(不涉及任何硬件的代码)时，图形界面是软件运行的最简单直接的体现，也使用户操作方便，简单易懂。本章的程序代码更深一层次地向读者演示了 Linux 下 GTK+图形界面编程库中相关函数的调用方法，建议读者可阅读光盘中的源代码。

第16章

Linux远程管理工具的设计

现如今，Linux 已成为应用比较广泛的操作系统之一，在其系统中有很多的服务。为了实现远程配置和管理 Linux 中的各种服务，需要有一种远程管理 Linux 系统的工具软件，Webmin 就是一种较好的 B/S 模式软件。本章将带领读者自行设计一个基于 C/S 模式实现的类似于 Webmin 的 Linux 系统远程管理工具。该工具的主要功能是实现对 Linux 系统用户和组的添加、修改和删除；对系统中的应用服务(如：DNS、FTP、Apache、系统启动服务管理)进行管理和配置，这些服务的远程配置主要是通过修改相应的服务配置文本文件来实现的。该远程管理工具选用 C/S 模式设计，在客户端，采用 GTK+图形编程来实现操作界面；在服务器端，选用 Linux 系统的文件调用函数来读写配置文件中的数据；采用基于 TCP 的 Socket 编程来实现客户端和服务器端之间的数据通信。



本章内容：

◎ 软件功能的概述。

◎ 服务器端程序的设计。

◎ 客户端程序。

16.1 软件功能概述

在介绍软件的具体实现流程之前，需要先对软件的功能有大致的了解。本章所要介绍的 Linux 系统远程管理工具仍然分为客户端和服务器端两个部分。客户端的主要功能是用户图形界面的实现及操作界面时相应信号与事件的产生，服务器端的主要功能则是接收客户端发送来的数据，以及根据这些数据修改 Linux 系统下的相应配置文件，从而实现了客户端到服务器端的远程管理。

16.1.1 Webmin 简介

Webmin 是目前功能最强大的基于 Web 的 UNIX 系统管理工具。管理员通过浏览器访问 Webmin 的各种管理功能并完成相应的管理动作。目前，Webmin 支持绝大多数的 UNIX 系统，这些系统除了各种版本的 Linux 以外还包括 AIX、HPUX、Solaris、Unixware、Irix 和 FreeBSD 等。

在 Webmin 的管理界面，借助任何支持表格和表单的浏览器和 File Manager 模块所需要的 Java，Linux 系统管理员就可以设置远程主机的用户账号、Apache、DNS、文件共享，等等。Webmin 包括一个简单的 Web 服务器和许多 CGI(Common Gateway Interface，公共网关接口)程序，这些程序可以直接修改系统配置文件，比如/etc/inetd.conf 和/etc/passwd。Web 服务器和所有的 CGI 程序都是用 Perl 5 语言编写的，没有使用任何非标准的 Perl 模块。

Webmin 让 Linux 用户能够在远程使用支持 HTTPS(Secure Hypertext Transfer Protocol，安全超文本传输协议)协议的 Web 浏览器通过 Web 界面管理自己的主机。Webmin 工具在保证安全性的前提下提供了简单深入的远程管理，这使得 Webmin 对于 Linux 系统管理员来说是非常理想的，因为所有主流平台都有满足甚至超出上述需求的 Web 浏览器。而且 Webmin 有其自己的“Web 服务器”，因此不需要运行第三方软件(比如 Web 服务器)。

同时，Webmin 是可扩展的，Webmin 的模块化架构允许用户在需要时编写自己的配置模块，以使 Webmin 永远可以按照管理员的需要进行调整。Webmin 包括了许多模块，尽管目前我们将主要关注网络服务，但是对于一个 Linux 系统管理员来说，Linux 系统的每一部分都能够通过 Webmin 来配置和管理。Webmin 的另一个可以看成其简化版本的主要针对普通用户的软件就是 Usermin。

下面将向读者讲解如何自行设计一个类似于 Webmin 的 Linux 系统远程管理工具。在本书附带的光盘中给出了该软件的全部源代码。

16.1.2 软件总体设计

本章所要介绍的 Linux 系统远程管理工具是基于 C/S 模式的实现。客户端程序的主要功能是用户图形界面的编写，以使用户端的操作方便易懂。另外，客户端负责和服务器端程序进行通信，主要是向服务器发送用户的配置类型及配置数据。客户端的图形用户界面采用 GTK+程序编写，客户端与服务器间的数据通信采用基于 TCP 协议的 Socket 编程来实现。

服务器端程序的主要功能是接收客户端程序的数据，并且通过收到的数据来完成系统或应用服务程序相应文件的配置与修改。在 Linux 中，系统服务配置文件主要是以文本文件的形式

存在的，所以通过 Linux 的相关系统调用(这些函数已经在本书的第 7 章向读者介绍)很容易修改这些配置文件。软件的总体设计流程如图 16.1 所示。

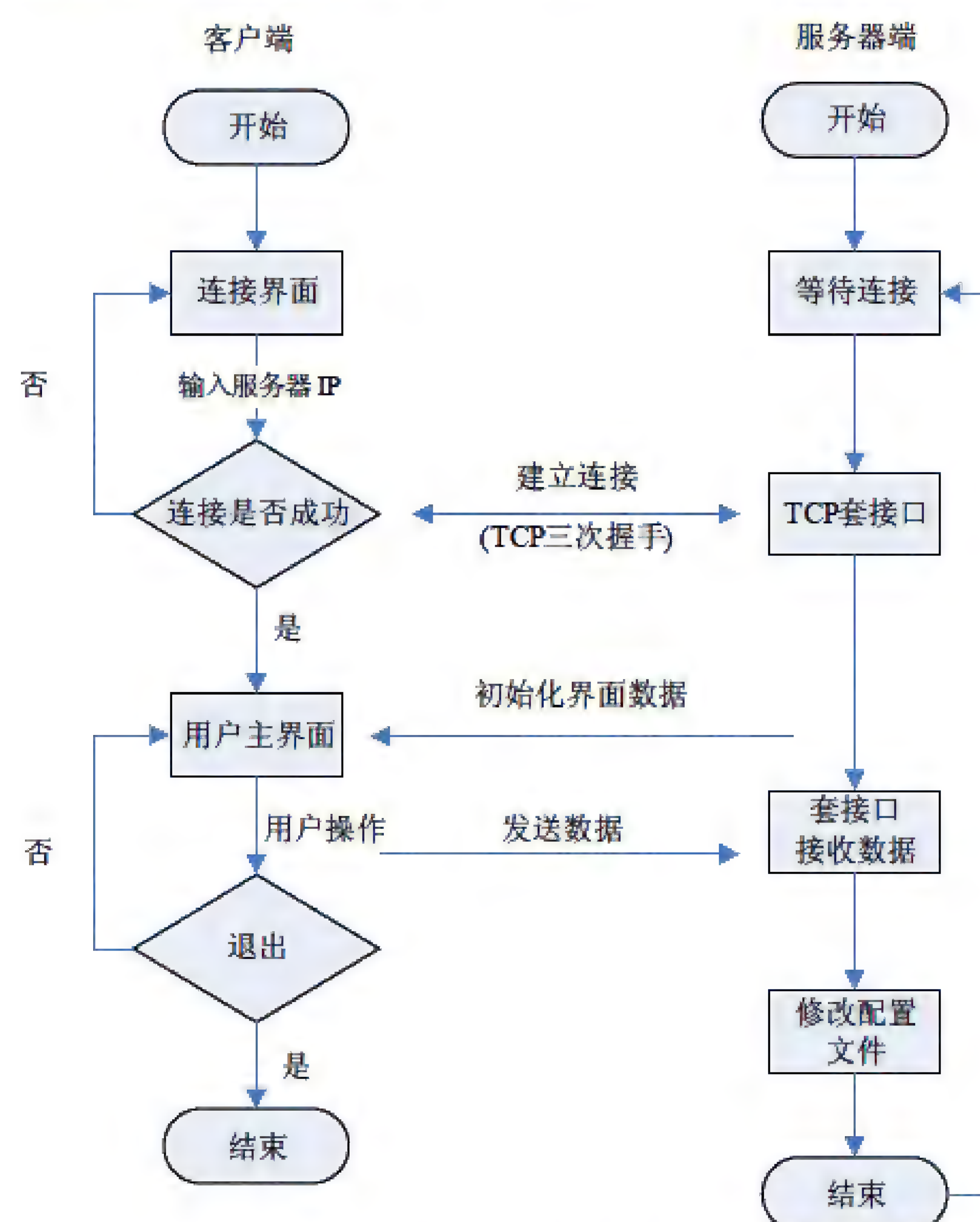


图 16.1 软件总体流程图

客户端的程序主要包括连接界面和用户操作的主界面，为用户的图形界面提供了接口。

服务器端的程序包括 Linux 系统用户管理操作、用户组的管理操作、系统服务启动管理、DNS 管理操作、Apache 服务管理操作和 FTP 服务管理操作等。服务器端各个模块的主要功能概述如下：

- 用户管理操作模块：实现对 Linux 系统用户的添加、删除、浏览和修改操作。
- 用户组管理操作模块：实现对 Linux 系统用户组的添加、删除、浏览和修改操作。
- 系统服务启动管理模块：实现对 Linux 系统下的某些服务的管理，如 http、sendmail、smb 服务等。
- DNS 管理操作模块：实现对 Linux 系统下 DNS 服务的管理，包括添加、删除区域信息，添加、删除区域中的域名。
- Apache 服务管理操作模块：实现对 Linux 系统下 Apache 服务的管理，包括 Apache 服务的停止、启动和重启，读取配置文件内容，修改配置文件，查看服务错误日志等操作。
- FTP 服务管理操作模块：实现对 Linux 系统下 FTP 服务的管理，包括 FTP 服务的停止、启动和重启，读取配置文件内容，修改配置文件内容等操作。

16.2 服务器端程序设计

前面已向读者提到，服务器端的程序包括多个功能模块，下面简要向读者介绍软件的设计

流程，具体的代码可参见本书附带光盘中的内容。

16.2.1　服务器端工作流程

服务器程序的主要功能有两个，一是监听 TCP 套接字端口，接收客户机发送来的数据，以及向客户机发送数据；二是根据客户程序发送过来的数据对相应的配置文件进行修改。服务器端的工作流程图如图 16.2 所示。

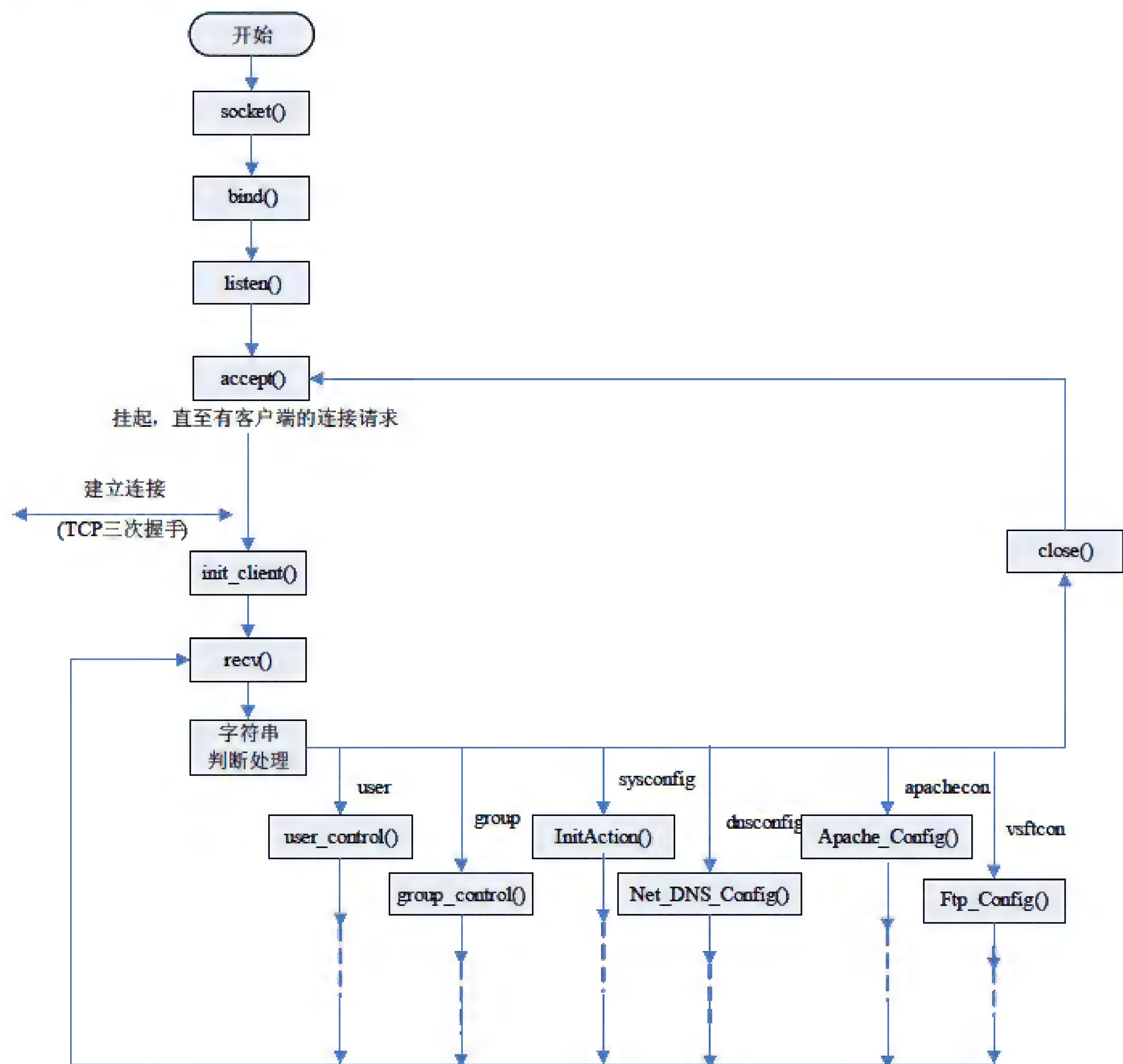


图 16.2　服务器端工作流程图

在图中清楚地显示了服务器端程序的工作流程，首先是服务器建立 Socket(这在第 12 章已经详细向读者讲解了)，接着是套接字监听端口，挂起(或阻塞)于 accept()调用，等待客户机程序的连接请求。当成功建立与某一客户机的连接后，init_clinet()函数从配置文件中读取服务器的当前配置文件数据，然后发送给客户端程序，让客户端程序通过这些数据来初始化界面。此时用户便可以通过客户端的界面操作进行系统的配置与管理了，客户端程序将用户修改的配置数据以字符串的形式发送给服务器。服务器接收客户程序的数据(流程控制字符串)，通过判断字符串的类型来控制服务器程序执行不同的流程(图 16.2 中的虚线部分)。

从图 16.2 中可以看到，服务器端针对不同的字符串，会进行不同的配置操作，即修改不同 Linux 系统或应用程序的配置文件。这些不同的配置操作可以理解为服务器端的不同功能模块，下面分别对这些模块加以介绍。

16.2.2　系统用户管理操作

当服务器和客户端建立了连接，并且收到了客户端程序发送来的“user”字符串后，服务

器端程序便进入系统用户管理操作模块，接着接收客户端程序发送过来的数据(字符串)，通过数据来判断是对用户的添加、删除、浏览或修改操作。如果接收到了“adduser”字符串就进入添加用户流程；接收到“deluser”字符串就进入删除用户流程；接收到“scanuser”字符串就进入浏览用户流程；接收到“property”字符串就进入修改用户流程；接收到“exit”字符串则退出。系统用户管理操作的流程如图 16.3 所示。

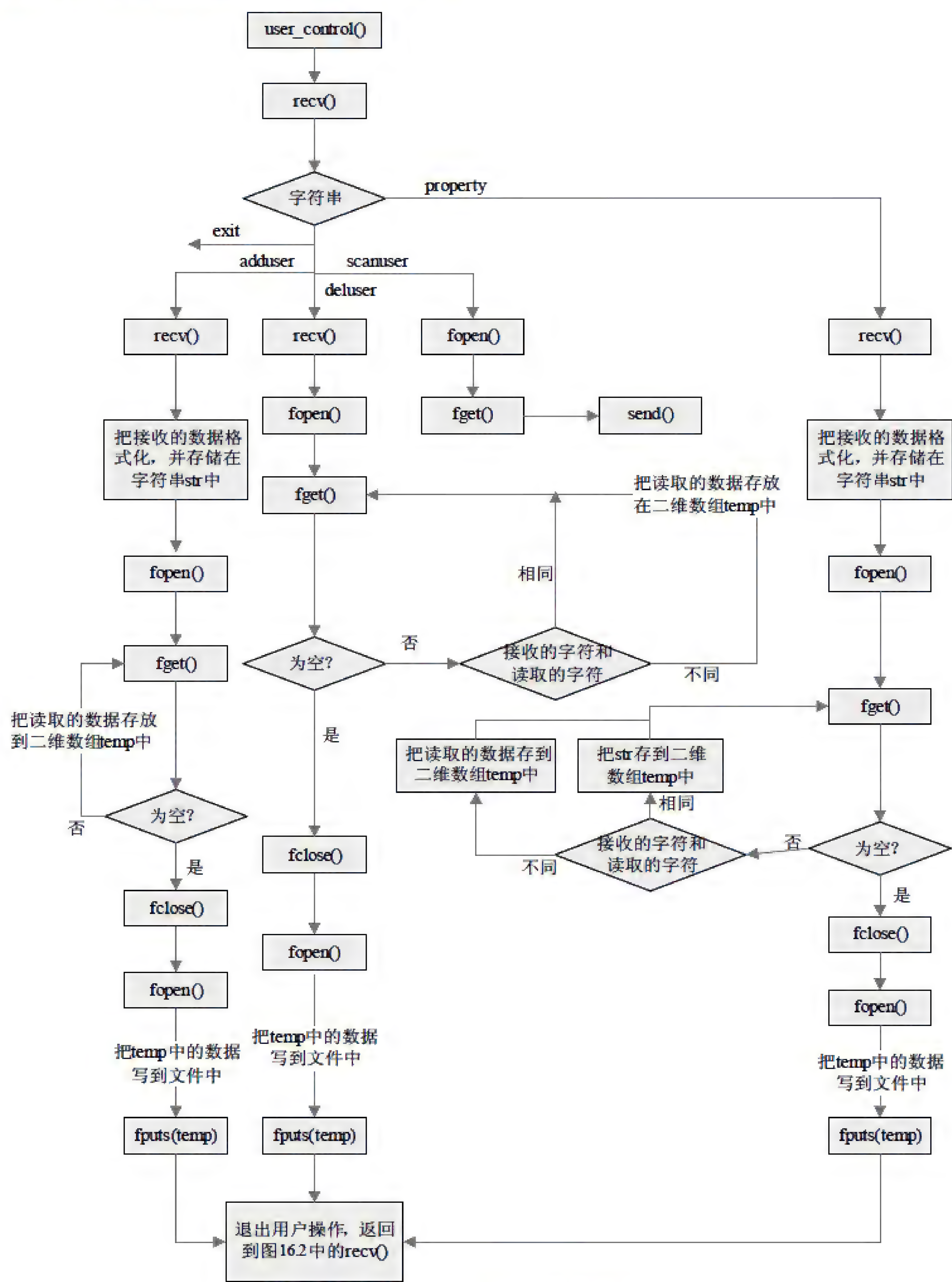


图 16.3 用户管理操作流程

对用户的管理操作是指添加、删除或修改 Linux 的当前用户。Linux 用户都是记录在/etc/passwd 文件中的，对用户添加、删除或修改的操作都是通过修改/etc/passwd 配置文件来实现的。/etc/passwd 中记录的用户都是按照统一格式来存储的，格式举例如下：

```
zhangfan:x:500:500:zhangfan:/home/zhangfan:/bin/bash
```

可以看到，每一个记录由多个字段组成，各个字段之间用“:”隔开。各个字段的含义分别如下：

- 第一字段：普通用户的用户名。在上面的例子中，我们看到用户的用户名是 zhangfan。
- 第二字段：口令，在例子中我们看到的是一个 x，其实密码已被加密并映射到/etc/shadow 文件中。
- 第三字段：用户的 ID。
- 第四字段：用户组的 ID。
- 第五字段：用户名全称，可选项。
- 第六字段：用户的主目录所在位置。该用户的主目录是/home/zhangfan。
- 第七字段：用户所用 Shell 的类型，一般设置为/bin/bash。

提 示

读者可以使用“cat /etc/passwd”命令来查看自己的 Linux 主机中用户管理配置文件的内容。

程序进入到添加用户的流程后，服务器接收客户程序传过来的将要添加的用户信息(包括用户名、密码、全名)。然后对客户程序发送过来的用户密码用 MD5 算法加密，并将加密后的密码和用户其他数据按照/etc/passwd 的统一字符串格式复制到字符串变量 str 中。用基于流的 I/O 操作并以只读方式打开文件/etc/passwd(参考第 7 章)，读取/etc/passwd 中的每一行，把每次读取的数据都复制到二维数组 temp 中，读取完毕后关闭文件。再以写的形式打开文件，然后把上面二维数组 temp 中的内容写到文件中，最后再把变量 str 的内容写到文件中，这样就实现了用户的添加。

程序进入到删除用户流程后，服务器接收客户程序传过来的将要删除用户的用户名。同样以只读方式打开文件/etc/passwd，并依次读取文件中的数据。判断读取的数据，当要删除的用户名和读取的用户名一致时，就不把这个用户信息复制到二维数组 temp 中；当删除用户名和读取用户名不一致时，就把读取数据复制到 temp 中。读取完毕后关闭这个流。重新以写文件的操作方式打开文件/etc/passwd，然后把二维数组 temp 中的数据以流的形式再重新写入到文件/etc/passwd 中，这样就实现了用户的删除。

当程序进入用户修改的流程时，服务器接收客户程序发送过来的数据后，把接收到的数据格式化复制到数组 str 中。同样以只读方式打开文件/etc/passwd，并依次读取文件中的数据。判断读取的数据，当要修改的用户名和读取的用户名一致时，就把数组 str 中的数据复制到二维数组 temp 中；当修改的用户名和读取的用户名不一致时，就把读取的数据复制到 temp 中。读取完毕后关闭这个流。重新以写文件的操作方式打开文件/etc/passwd，然后把二维数组 temp 中的数据以流的形式再重新写入到文件/etc/passwd 中，这样就完成了修改用户的操作。

当进入浏览用户的流程后，服务器打开/etc/passwd 文件，把文件的数据读取到数组 buff 中，读取完毕后关闭文件，并将 buff 中的数据发送到客户端，客户端程序接收到数据后显示当前的用户信息。

16.2.3 系统用户组的操作

同用户管理的操作类似，当服务器收到了客户端程序发送来的“group”字符串后，服务

器端程序便进入系统用户组的管理操作模块，接着接收客户端程序发送过来的数据(字符串)，通过数据来判断是对用户组的添加、删除、浏览或是修改操作。如果接收到“groupadd”字符串就进入添加用户组的流程；接收到“groupdel”字符串就进入删除用户组的流程；接收到“groupscan”字符串就进入浏览用户组的流程；接收到“property”字符串就进入修改用户组的流程；接收到“exit”字符串则退出。系统用户组管理操作的流程如图 16.4 所示。

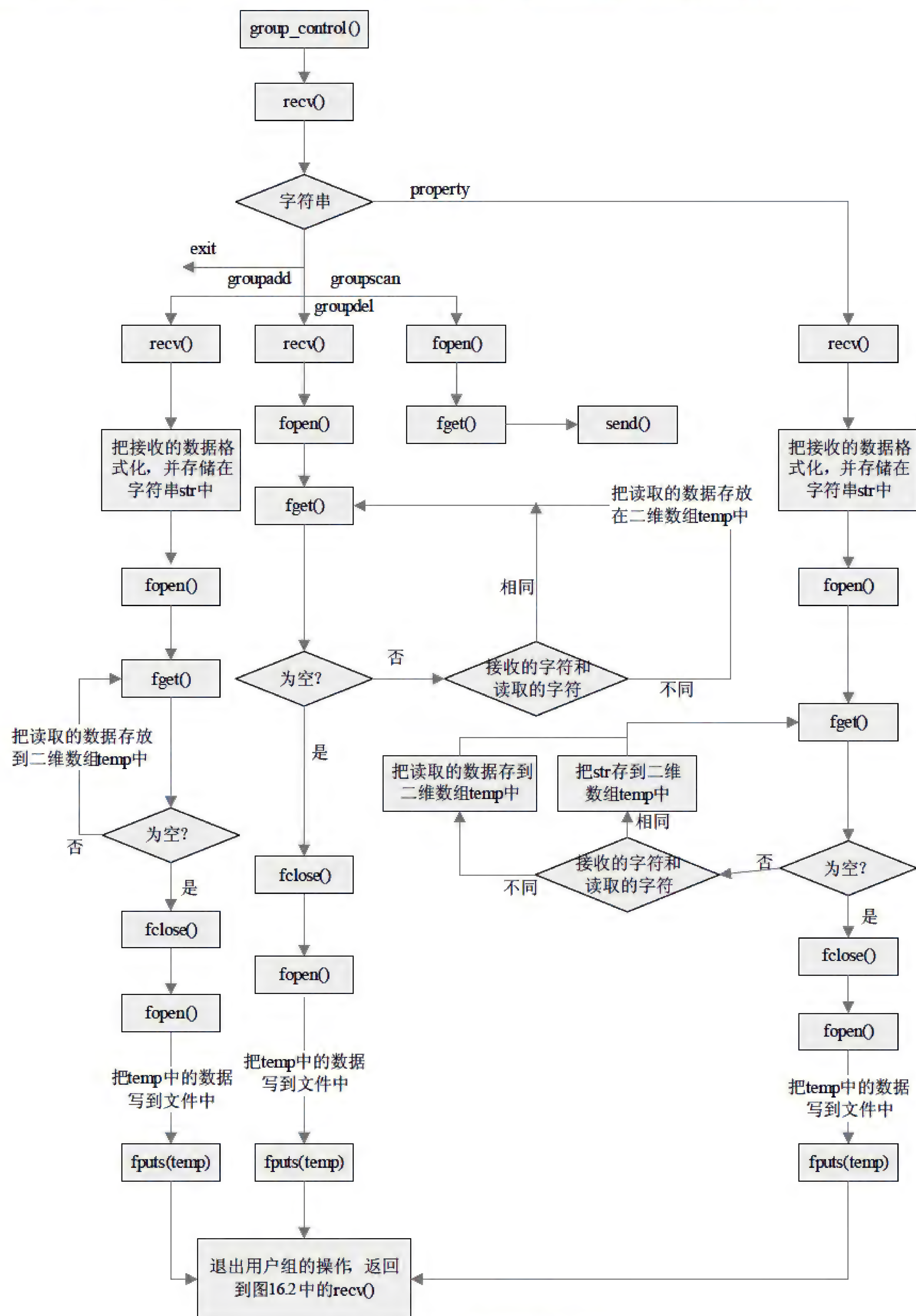


图 16.4　用户组管理操作流程

Linux 系统下，用户组(Group)的配置文件主要有/etc/group 和/etc/gshadow，其中/etc/gshadow 是/etc/group 的加密信息文件。etc/group 文件是 Linux 系统用户组的配置文件，内容包括用户和用户组，并且能显示出用户是归属哪个用户组或哪几个用户组，因为在 Linux 中，一个用户可以归属一个或多个不同的用户组；同一用户组的用户之间具有相似的特征。对组的操作都是通过修改/etc/group 文件的内容来实现的，主要是对组的添加、删除和修改。

与用户名信息配置文件/etc/passwd 类似，/etc/group 中记录的用户组的信息也是按照统一格式来存储的，比如：

```
zhangfan:x:500:
```

每一行的字串共由 4 个字段构成，各个字段之间也是用“:”隔开。各个字段的含义分别如下：

- 第一字段：用户名。在上面的例子中，我们看到用户的用户名是 zhangfan。
- 第二字段：口令，此处的密码已被加密映射到/etc/gshadow 文件中。
- 第三字段：用户组的 ID。
- 第四字段：组成员名。可以把需加入该组的用户名以逗号分隔添加到这里，同一组的成员可继承该组所拥有的所有权限。上面的例子中，组成员暂为空。

提 示

读者可以使用“cat /etc/group”命令来查看自己的 Linux 主机中用户组管理配置文件的内容。

如图 16.4 所示，当程序进入到组的添加流程后，服务器接收客户端发送过来的将要添加的组的信息，然后把接收的数据按照上述/etc/group 文件中的存储格式格式化后，复制到数组 str 中。接着以只读方式打开文件/etc/group，读取文件中组的信息数据，复制到二维数组 temp 中，读取完毕后关闭文件。然后以只写方式打开文件/etc/group，将 temp 中的数据写到文件中去，最后再把 str 中的内容也写入到文件中，关闭文件，这样就实现了组的添加。

程序进入到组的删除流程后，服务器接收客户程序发送过来的数据，同样以只读方式打开文件/etc/group，并依次读取文件中的数据。判断读取的数据，当读取的数据和发送过来的组名一致时，就不把读取的数据复制到二维数组 temp 中，其他情况则将读取的数据复制到 temp 中，读取完毕后关闭文件。再次打开文件/etc/group，把二维数组 temp 中的数据都写到文件中去，关闭文件，这样就完成了组的删除操作。

程序进入到组的修改过程，服务器接收客户端发送过来将要添加的组的信息，然后把接收的数据格式按照/etc/group 存储格式格式化后，复制到数组变量 str 中。接着打开/etc/group 文件，读取文件的数据，当读取的数据和发送过来的组名一致时，就把 str 的内容复制到 temp 中，其他情况则将读取的数据复制到 temp 中，读取完毕后关闭文件。再次打开文件/etc/group，把 temp 的数据都写到文件中去，关闭文件，这样就完成组的修改操作。

程序进入到浏览组信息的过程中，服务器打开/etc/group 文件，把文件中的数据读取到数组 buff 中，读取完毕后关闭文件，并将 buff 中的数据发送到客户端。客户端程序显示当前的用户组信息。

16.2.4　系统服务启动管理

系统服务启动的管理和其他的操作不大一样，其他的操作都是通过修改相应配置文件来实现的，而系统服务的启动则是通过添加和删除链接文件来完成的。其流程如图 16.5 所示。

系统服务启动管理的工作流程可描述如下：

当服务器端的程序进入到图 16.2 中的 InitAction()函数时，服务器便进入了系统服务启动管理流程。服务器接收客户端程序发送过来的数据，这里发送过来的是一个有 5 个字符的字符串，依次判断这 5 个字符。对于第 1 个字符，如果是 1，就调用 Linux 系统调用 link()将/etc/init.d/named shell 脚本文件链接为/etc/rc.d/rc5.d/S11named 文件；如果字符是 0，就调用系统调用 unlink()取消/etc/rc.d/rc5.d/S11named 文件的链接。然后再判断第 2 个字符，如果是 1，就把/etc/init.d/http shell 脚本文件链接为/etc/rc.d/rc5.d/S85httpd 文件；如果第 2 个字符是 0，就取消文件/etc/rc.d/rc5.d/S85httpd 的链接。然后判断第 3 个字符，是 1 就调用系统 link()函数将/etc/init.d/sendmail 链接到/etc/rc.d/rc5.d/S80sendmail，是 0 就取消对/etc/rc.d/rc5.d/S80sendmail 的链接。然后判断第 4 个字符的值，如果是 1，就把/etc/init.d/dhcp shell 脚本链接为文件/etc/rc.d/rc5.d/S65dhcpd，是 0 则取消对/etc/rc.d/rc5.d/S65dhcpd 的链接。最后再判断第 5 个字符，是 1 就把文件/etc/init.d/smb 链接为文件/etc/rc.d/rc5.d/S91smb，否则就取消对/etc/rc.d/rc5.d/S91smb 的链接。

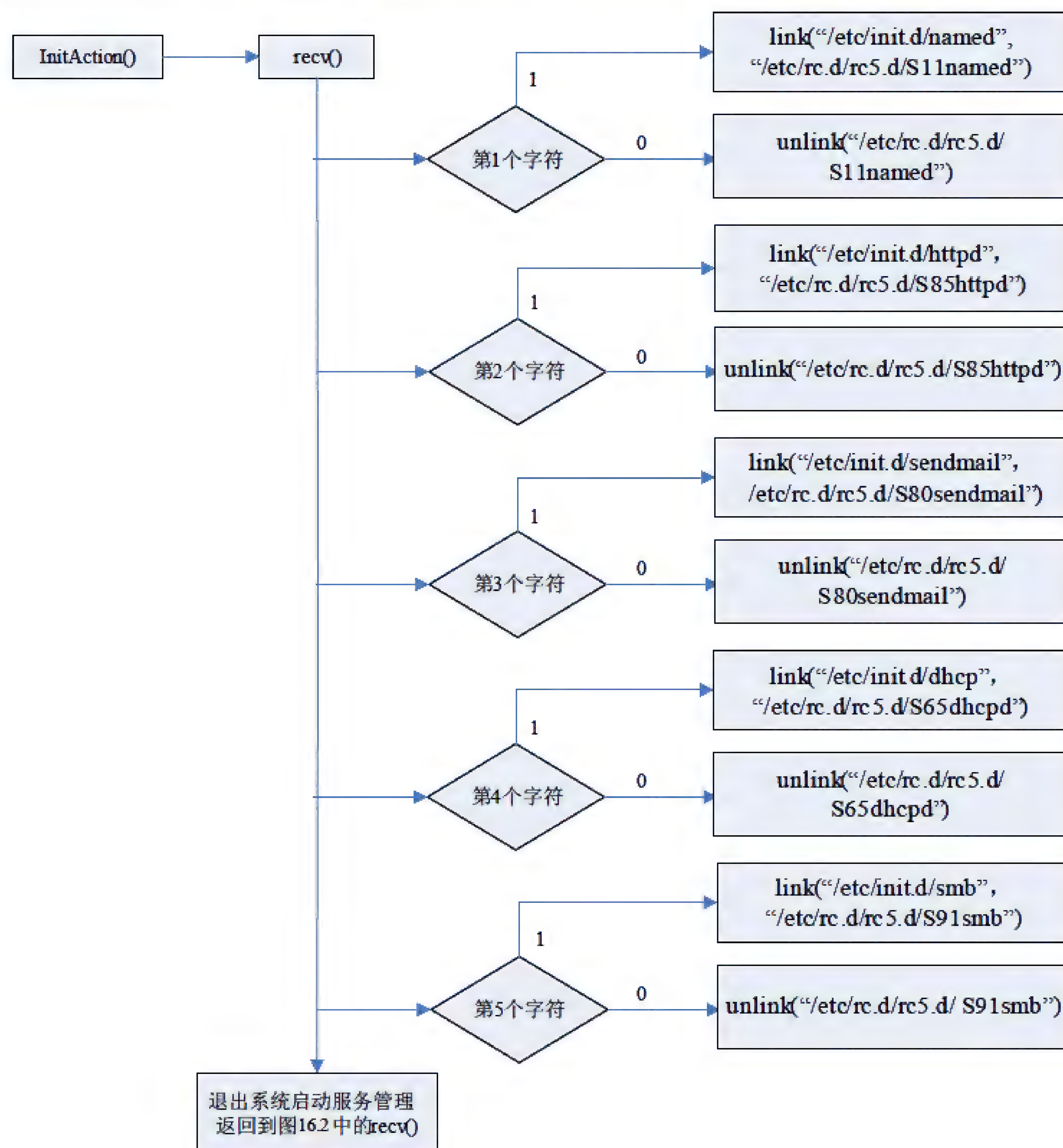


图 16.5　系统服务启动管理

提 示

对于 Linux 的链接文件的操作函数 link()和 unlink()，在 6.5.2 小节中已向读者详细讲解了。

16.2.5 DNS 管理操作

DNS 是 Domain Name System(域名系统)的缩写，域名是由圆点分开的一串单词或字母缩写组成的，每一个域名都对应一个唯一的 IP 地址，这种命名的方法或这样管理域名的系统叫作域名管理系统(参考 12.3.1 小节中的例子)。域名虽然便于人们记忆，但网络中的计算机之间只能互相识别 IP 地址，它们之间的转换工作称为域名解析(如 12.3.1 节中的域名 www.baidu.com 与 IP 地址 119.75.213.61 之间的转换)，域名解析需要由专门的域名解析服务器来完成，DNS 就是进行域名解析的服务器。

当服务器端的程序进入到图 16.2 中的 Net_DNS_Config()函数时，服务器便进入了 DNS 管理操作功能模块。该模块主要是在系统的 DNS 服务配置文件(Red Hat Linux 9.0 系统下为/etc/named.conf 文件)中添加或删除 zone(区域)信息，以及在 zone 中添加和删除 domain(域名)信息。进入该模块后，服务器程序同样会接收一个客户端发送过来的字符串数据，该数据用来控制 DNS 模块的流程，其流程如图 16.6 所示。

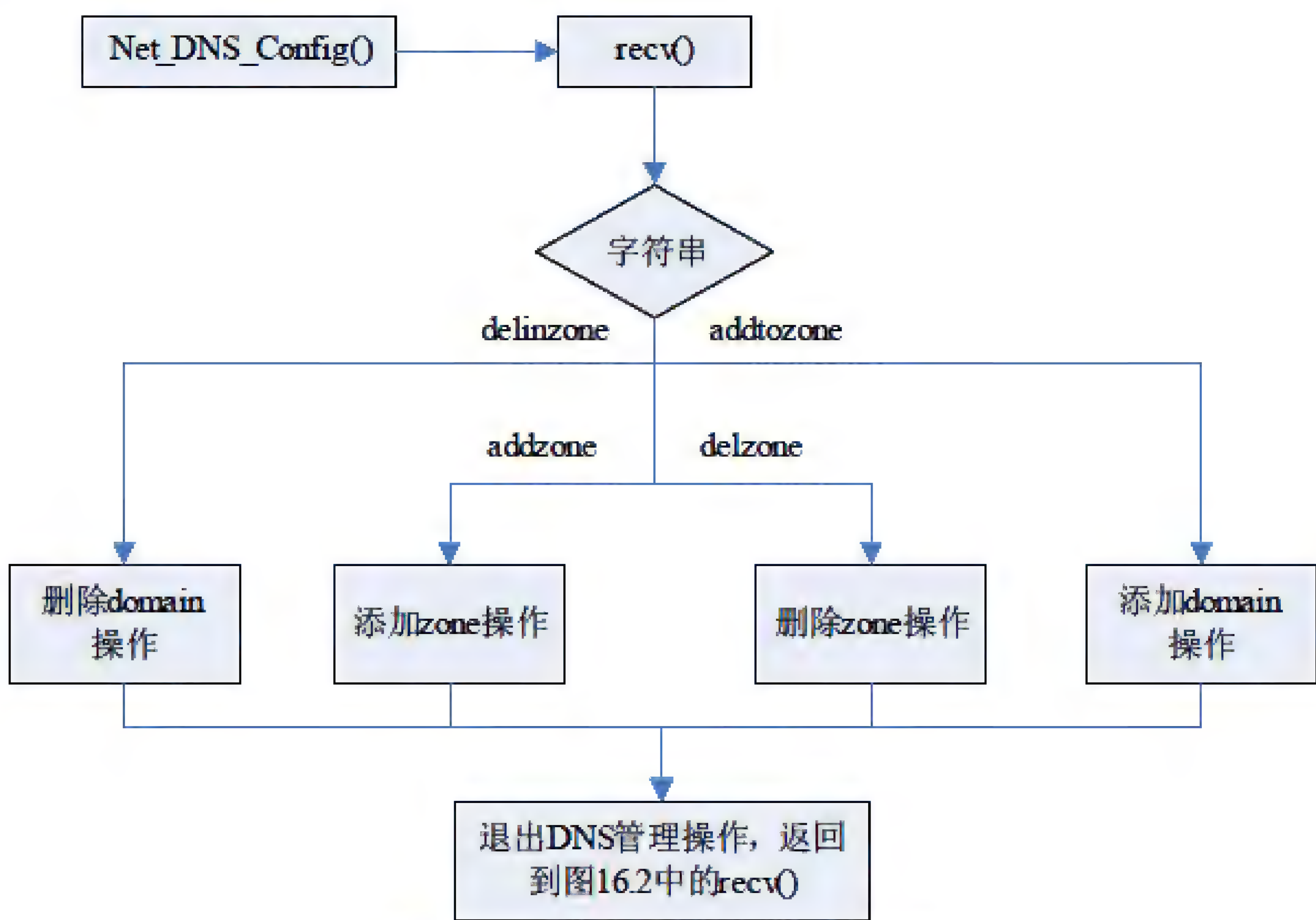


图 16.6 DNS 管理操作流程

如图 16.6 所示，当服务器接收到的字符串为“addzone”时，进入添加 zone 的操作流程。服务器接收客户端程序发送过来的数据(将要添加的 zone 名字和类型)，将 zone 的名字格式化，即加入点分隔符的格式后，判断系统中的 DNS 服务配置文件中是否已存在相同的区域 zone。如果存在，服务器便发送“zone name is exist”字符串到客户端，提示用户当前的区域 zone 已存在，并退出 DNS 配置操作；若不存在则再把 zone 的名字和类型格式化为配置文件/etc/named.conf 中 zone 的存储格式，然后把格式化后的数据存储到配置文件中，存储成功后发送“zone add succseed”字符串数据到客户端，提示用户添加区域成功。

当服务器接收到“delzone”字符串时，进入删除区域 zone 的操作流程。服务器接收客户

程序发送过来的数据(将要删除的 zone 名)，判断 DNS 服务配置文件中是否有相同的区域 zone 存在，如果没有就向客户端发送“zone name is not exist”字符串数据，提示用户重新输入；如果有就读取配置文件中的全部数据到数组 buff 中，再将 buff 中的内容与接收数据缓冲区的内容进行比较，相同则删除，不同则重新写回到文件中。写入成功后发送“zone delete succseed”字符串到客户端程序，提示用户删除区域成功。

当服务器接收到的字符串为“addtozone”时，进入添加域名 domian 的操作流程。服务器接收客户程序发送过来的数据(将要添加的域名 domian、IP 和区域名 zone)，然后判断 DNS 配置文件/etc/named.conf 中是否有区域 zone 存在，如果不存在，就发送“zone name is not exist”到客户端，提示用户重新输入；如果存在，则得到其数据文件的路径名，然后把 domian 和 IP 格式化后写入到相应的数据文件中去。

当服务器接收到的字符串为“delinzone”时，进入删除域名 domain 的操作流程。服务器接收客户程序发送的数据(将要删除的域名 domian 和这个域名所在的区域名 zone)，然后判断/etc/named.conf 文件中是否有这个 zone 存在，如果没有就发送“zone name is not exist”到客户程序，如果有就得到其数据文件名字，然后在数据文件中查看是否有 domain 存在，如果没有就发送“Domain name is not exist”到客户程序，如果有就删除 domain 部分，然后发送“Domain del succseed”到客户程序。

16.2.6 Apache 服务管理操作

在介绍 Apache 服务的管理操作之前，有必要先向读者介绍 Apache 服务软件。同时，为了试验本小节所讲述的程序，读者应该在自己的 Linux 主机中先安装和配置好 Apache 服务软件。

1. Apache 简介

随着网络技术的普及、应用和 Web 技术的不断完善，Web 服务已经成为互联网上重要的服务形式之一。原有的客户端/服务器模式正在逐渐被浏览器/服务器模式所取代。Apache 是目前最流行、使用最多的一款开放源代码的 Web 服务器软件。

统计显示，最流行的 Web 服务器是 OSS/FS。比如，Apache 就是现在排行第一的 Web 服务器，其市场份额比位于第二位的 IIS 高出了一倍多。Apache 的市场占有率表现出几个使对方望尘莫及的优势：

- 起源于 HTTP 协议——降低了用户加入协议来支援新的应用软件的门槛。
- 给 UNIX/Linux 带来生机——Apache 走到哪里，UNIX/Linux 就走到哪里。
- 支援厂商(如 IBM)的支持，为 Apache 提供的工具/模块持续成长。

下面简要概述 Apache 的工作原理。

Web 系统是客户端/服务器式的，所以应该有服务器程序和客户端程序两部分。常用的服务器程序是 Apache；常用的客户端程序是浏览器(如 IE、Netscape、Mozilla)。我们可以在浏览器的地址栏内输入统一资源定位地址(URL)来访问 Web 页面。Web 最基本的概念是超文本(Hypertext)。它使得文本不再是传统的书页式文本，而是可以在阅读过程中从一个页面位置跳转到另一个页面位置。用来书写 Web 页面的语言称为超文本标记语言，即 HTML。WWW 服务遵从 HTTP 协议，默认的 TCP/IP 端口是 80，客户端与服务器的通信过程简述如下：

(1) 客户端(浏览器)和 Web 服务器建立 TCP 连接，连接建立以后，向 Web 服务器发出访

问请求(如 get)。根据 HTTP 协议，该请求中包含了客户端的 IP 地址、浏览器的类型和请求的 URL 等一系列信息。

(2) Web 服务器收到请求后，将客户端要求的页面内容返回到客户端。如果出现错误，那么返回错误代码。

(3) 断开与远端 Web 服务器的连接。下面是一个客户端发送给 Web 服务器请求的数据包的内容：

```
GET /engineer/ideal/list.htm HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
application/vnd.ms-powerpoint, application/vnd.ms-excel, application/msword, */*
Referer: http://www.linuxar.com.cn/engineer/ideal/
Accept-Language: zh-cn
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
Host: www.linuxar.com.cn
Connection: Keep-Alive
```

从代码中可以看到，在客户端的请求里包含了很多有用的信息，如客户端类型等。Web 服务器会将请求的 Web 页内容发送返回给客户端。HTTP/1.1 说明：HTTP/1.1(超文本链接协议 1.1 版本)是 HTTP 协议的最新版本。HTTP 协议是运行在 TCP/IP 协议组上的万维网应用协议。HTTP/1.1 提供了比前一版本更快的访问网站速度，同时针对网络资源进行优化，降低了网络流量。HTTP/1.1 由互联网工程任务组开发。现在大部分服务器和网站都支持 HTTP/1.1 协议。

下面是 HTTP/1.1 协议能够加快网页访问速度的原因。

(1) 以往的 HTTP 协议每次访问应用程序时，都会进行创立及撤销链接的步骤。而 HTTP/1.1 在首次访问网站时建立持久链接，将多个请求批量或通过管道发送到输出缓冲区内。TCP 协议允许将多个来自 IP 层的数据包请求或回复命令集中到一个 TCP 段中。因此减少了反复建立链接所需的时间，同时由于没有了不必要的申请链接数据包，也降低了网络流量。由于将命令通过管道输送，大大提高了 TCP 段的效率。总之，网络流量降低了，性能提高了。

(2) 当支持 HTTP/1.1 的浏览器发现网页是未压缩网页时，会将网页进行压缩后进行传输，这样可以节约更多流量空间，不过由于网页中的图片文件一般都已经被压缩过，因此，这种压缩对图片多的网页不太有效。除持久链接及其他改进后的性能之外，HTTP/1.1 还允许多个域名共享同一 IP 地址。这简化了网络服务器对虚拟主机数目管理的处理量。

另外，Apache 的主要特征如下：

(1) 支持 HTTP/1.1 协议。Apache 是最先使用 HTTP/1.1 协议的 Web 服务器之一，它完全兼容 HTTP/1.1 协议并与 HTTP/1.0 协议向后兼容。Apache 已为新协议所提供的全部内容做好了必要的准备。

(2) 支持通用网关接口(CGI)。Apache 用 mod_cgi 模块来支持 CGI，它遵守 CGI/1.1 标准并且提供了扩充的特征，如定制环境变量和很难在其他 Web 服务器中找到的调试支持功能。

(3) 支持 HTTP 认证。Apache 支持基于 Web 的基本认证，它还为支持基于消息摘要的认证做好了准备。Apache 通过使用标准的口令文件 DBM SQL 调用，或通过对外部认证程序的调用来实现基本的认证。

(4) 集成的 Perl 语言。Perl 已成为 CGI 脚本编程的基本标准。Apache 肯定是使 Perl 成为这

样流行的 CGI 编程语言的因素之一，现在 Apache 比以往任何时候都更加支持 Perl，通过使用它的 mod_perl 模块可以将基于 Perl 的 CGI 脚本装入内存，并可以根据需要多次重复使用该脚本。这消除了经常与解释性语言联系在一起的启动开销。

(5) 集成的代理 Proxy 服务器。Apache 可作为前向代理服务器也可作为后向代理服务器。

(6) 服务器的状态和可定制的日志。Apache 在记录日志和监视服务器本身状态方面提供了很大的灵活性，可以通过 Web 浏览器来监视服务器的状态，也可根据自己的需要来定制日志。

(7) 允许根据客户主机名或 IP 地址限制访问。

(8) 支持 CGI 脚本，如 Perl\PHP 等。

(9) 支持用户 Web 目录。Apache 允许主机上的用户使用特定的目录存放用户自己的主页。可以通过如下 URL 地址来访问，如用户 zhang，http://hostname~/zhang。

(10) 支持虚拟主机。即通过在一个机器上使用不同的主机名来提供多个 HTTP 服务。Apache 支持包括基于 IP、名字和 Port 3 种类型的虚拟主机服务。

(11) 支持动态共享对象。Apache 的模块可在运行时动态加载，这意味着这些模块可以被装入服务器进程空间，从而减少系统的内存开销。

(12) 支持服务器包含命令 SSI。Apache 提供扩展的服务器命令包含该项功能，为 Web 站点开发人员提供了更大的灵活性。

(13) 支持安全 Socket 层(SSL)。

(14) 用户会话过程的跟踪能力。通过使用 HTTP cookies，一个称为 mod_usertrack 的 Apache 模块可以在用户浏览 Apache Web 站点时对用户进行跟踪。

(15) 支持 FastCGI。Apache 使用 mod_fcgi 模块来实现 FastCGI 环境，并使 FastCGI 应用程序运行得更快。

(16) 支持 Java Servlets。Apache 的 mod_jserv 模块支持 Java Servlets 该项功能，可使 Apache 运行服务器的 Java 应用程序。

(17) 支持多进程。当负载增加时，服务器会快速生成子进程来处理，从而提高系统的响应能力。

2. Apache 服务管理操作

Apache 服务管理操作模块主要完成的功能是向客户程序发送当前的一些 Apache 服务错误日志；通过客户端发送过来的数据进行一些简单的配置；Apache 服务的开启、关闭和重启功能等，其流程如图 16.7 所示。

Apache 服务停止、启动和重启操作是通过 exec()函数调用 Apache 停止、启动和重启的 Shell 脚本来完成的。然后再读取环境变量，最后向客户程序发送操作结果。

发送配置文件内容操作是读取 Apache 配置文件的内容，然后把读取的内容发送到客户程序。

修改配置文件操作，先是接收客户程序发送过来的数据(要修改的内容)，然后在配置文件中查找到要修改的位置，并将客户程序发送过来的数据写到配置文件中。

查看错误日志操作，即读取错误日志中的数据，并把数据发送到客户端程序。

提 示

Apache 服务管理操作模块要求首先在 Linux 主机上安装了 Apache 服务器软件。可以从官方网站 http://www.apache.org/上下载 Apache 服务器的源代码及安装文件。关于 Apache 的安装与配置，读者可参考其他相关资料。

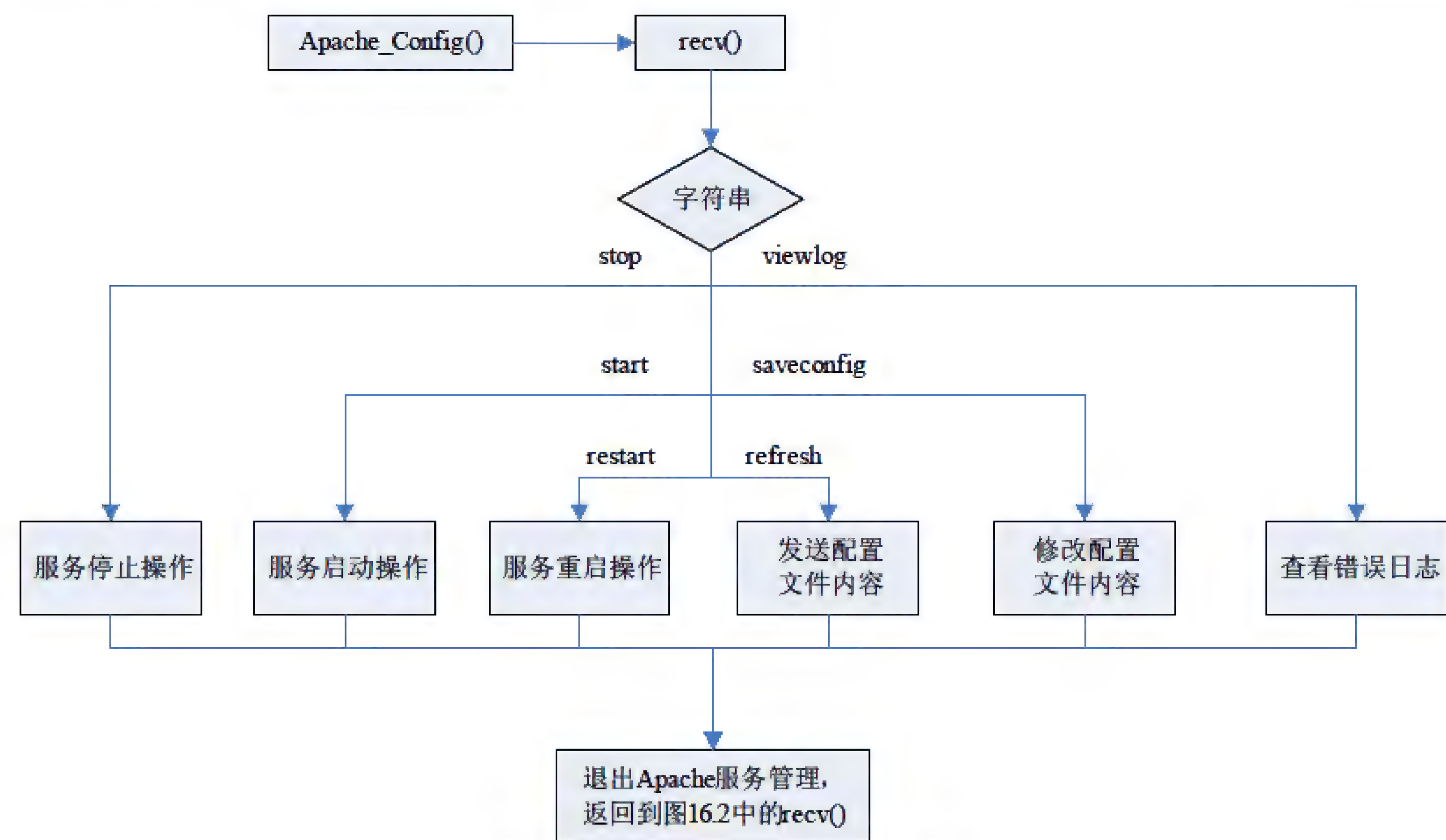


图 16.7 Apache 服务管理操作

16.2.7 FTP 服务管理操作

FTP 服务管理操作模块主要完成的是对 FTP 服务的停止、启动和重启，向客户端发送 FTP 配置文件的内容，以及通过客户程序发送过来的数据对 FTP 的配置文件进行简单的修改等。其流程如图 16.8 所示。

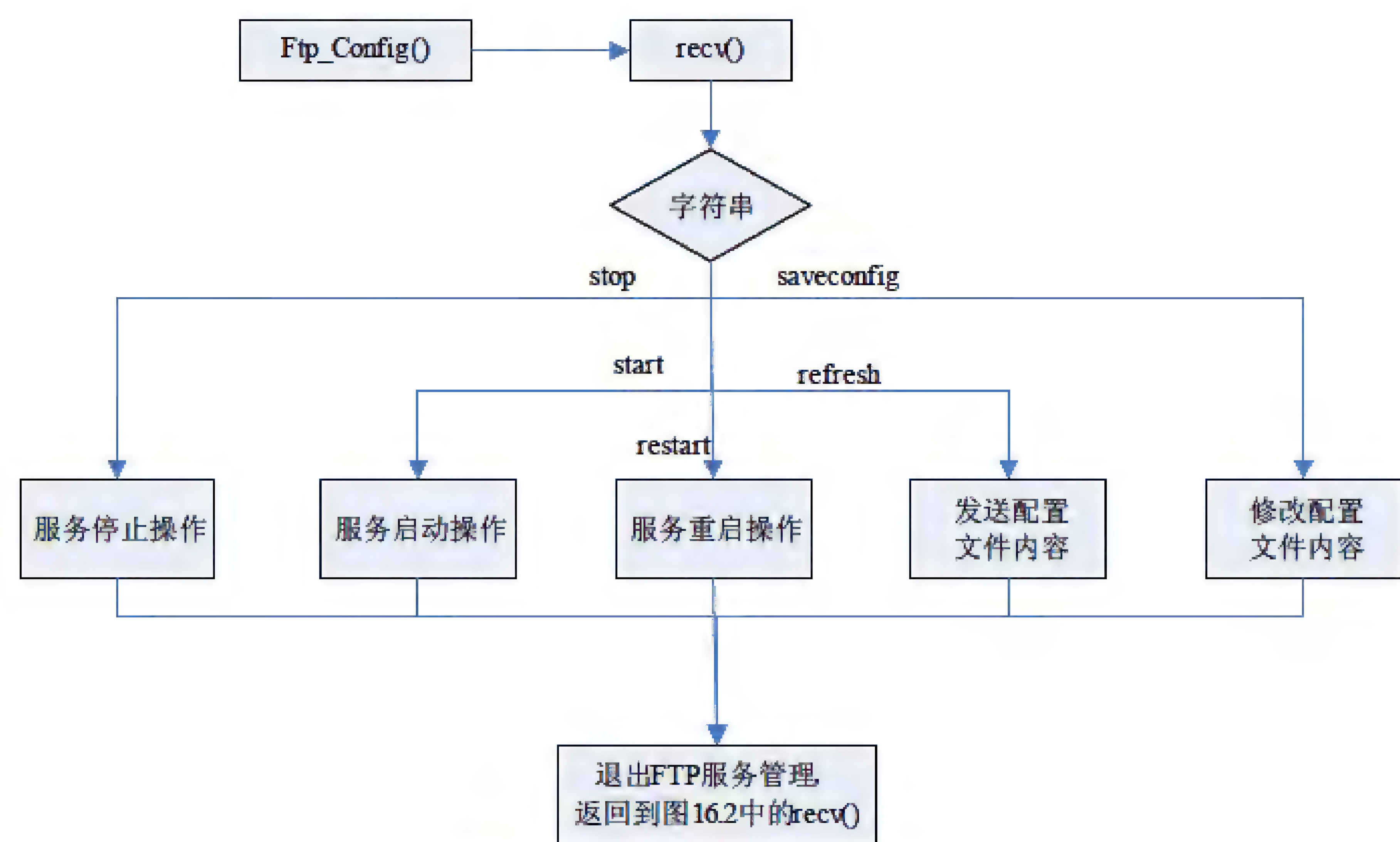


图 16.8 FTP 服务管理操作

同 Apache 服务的操作类似，FTP 服务的停止、启动和重启操作也是通过 exec()函数调用 FTP 停止、启动和重启的 Shell 脚本来完成的。然后读取环境变量，并向客户程序发送操作结果。

发送配置文件内容的操作是读取 FTP 配置文件的数据，然后把读取的内容发送到客户程序。

修改配置文件操作，先是接收客户程序发送过来的数据(要修改的内容)，在配置文件中找到要修改数据的位置，然后将收到的数据写入到文件中去。

16.3 客户端程序

客户端的作用有两个，一是为用户提供操作界面，二是和服务器端进行数据通信。

在用户界面方面，采用 GTK+编程来建立用户界面，把实现的功能接口通过界面的方式提供给用户，让用户来进行相关的操作。在和服务器通信方面，通过 Socket 编程来实现，把用户操作后的结果通过 Socket 来传送给服务器程序，服务器程序通过 Socket 传递的数据来对服务器端相应的系统服务配置文件进行配置和修改。相信这些对读者来说已经很熟悉了。

在模块划分方面，主要是通过功能来划分为用户的操作、组的操作、DNS 配置、Apache 的配置、FTP 的配置、启动服务管理。客户程序一共有两个窗口，连接窗口和主窗口。本节给出了客户端程序的设计流程及部分重要代码，详细的源代码读者可参见光盘中的内容。

16.3.1 连接界面

首先是连接界面的编写，主要是用于建立客户端和服务器端的连接。除了用 GTK+编写界面以外，还要用到 Socket 编程来进行和服务器端的 TCP 连接的建立。当用户输入服务器的 IP 地址，单击“connect”按钮后，便会建立与指定服务器的连接。连接界面是客户端程序的第一个界面，其流程图如图 16.9 所示。

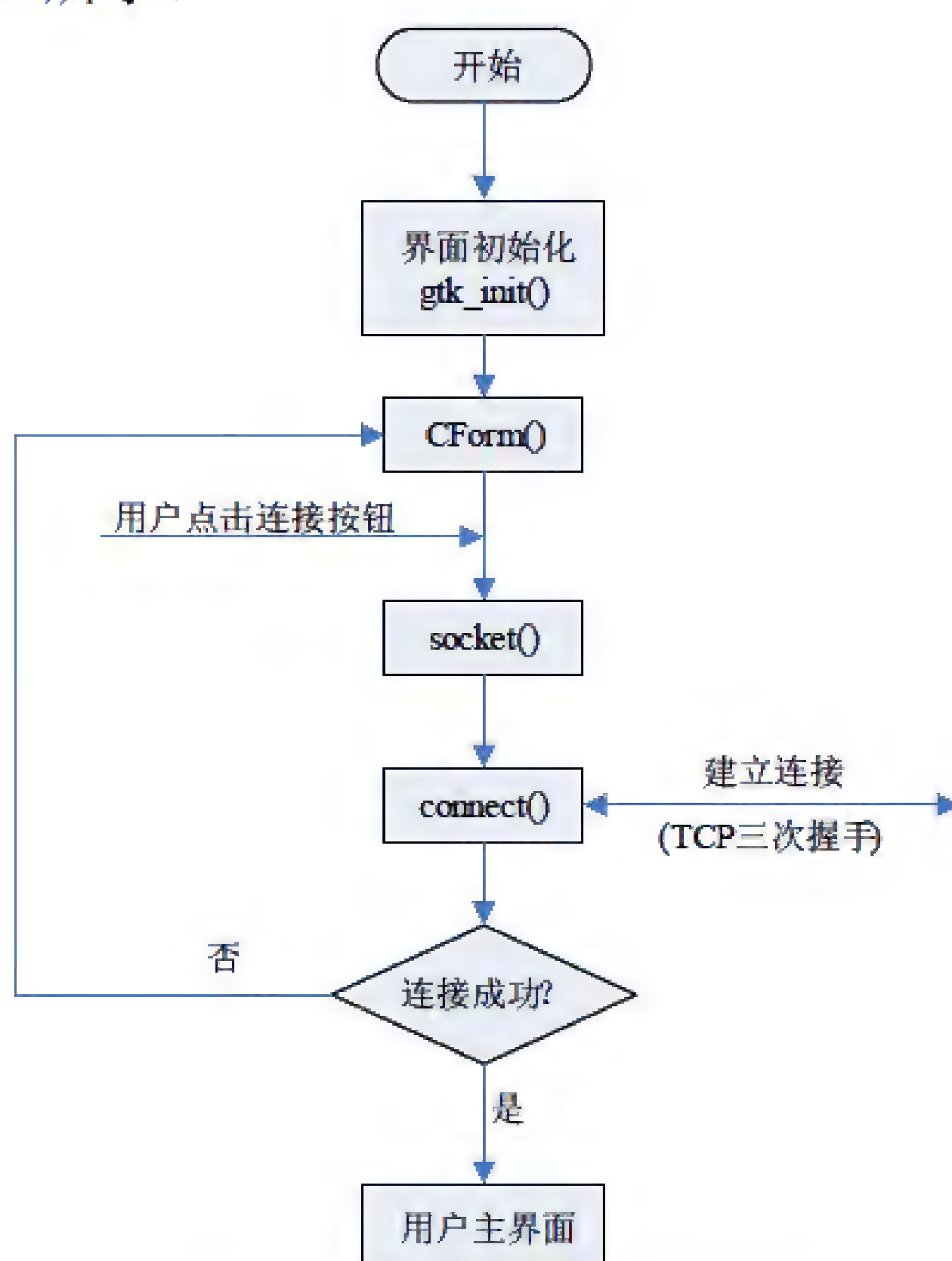


图 16.9　连接界面的流程图

程序从 main()进入以后就调用 gtk_init()函数。gtk_init()函数功能初始化 GTK+图形界面，并且分析在命令行中传递进来的参数。命令行中传递过来的任何参数，只要是它能识别的，都会从列表中删除，并且修改 argc 和 argv 的值，就像这些参数从不存在一样，然后应用程序分析剩余的参数。

然后进入 CForm()函数。这个函数的主要功能是创建连接界面，并且当用户在文本框中输入了服务器 IP 地址后，把文本框中的 IP 传递给后面的函数，实现与服务器网络连接。其函数代码如下(取自光盘的/src/chapter_16/clinet/conform.c 文件)：

```
GtkWidget *mainWin = NULL;
GtkWidget *textbox = NULL;
void CForm(int *sock)
{
GtkWidget *exitButt,*clloButt,*clloLabl;
    GtkWidget *table;
    int level;
    char *text;
    mainWin = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title((GtkWindow*)mainWin,"connet");
    gtk_window_set_default_size((GtkWindow*)mainWin,80,90);
    exitButt = gtk_button_new_with_label("exit");
    clloButt = gtk_button_new_with_label("connect");
    clloLabl = gtk_label_new("Servers IP");
    textbox = gtk_entry_new();
    table = gtk_table_new (3, 2, TRUE);
    gtk_table_set_row_spacing(GTK_TABLE(table),1,10);
    gtk_table_set_row_spacing(GTK_TABLE(table),0,10);
    gtk_table_set_col_spacing(GTK_TABLE(table),1,35);
    gtk_entry_set_text((GtkEntry *)textbox,"input severs IP here!");
    text = gtk_entry_get_text((GtkEntry *)textbox);
    gtk_table_attach_defaults((GtkTable*)table,clloLabl,0,1,0,1);
    gtk_table_attach_defaults((GtkTable*)table,textbox,1,2,0,1);
    gtk_table_attach_defaults((GtkTable*)table,clloButt,0,2,1,2);
    gtk_table_attach_defaults((GtkTable*)table,exitButt,0,2,2,3);
    gtk_container_set_border_width (GTK_CONTAINER (mainWin), 0);
    gtk_container_add (GTK_CONTAINER (mainWin), table);
    gtk_signal_connect(GTK_OBJECT(exitButt), "clicked",
GTK_SIGNAL_FUNC(Destroy),NULL);
    gtk_signal_connect(GTK_OBJECT(mainWin), "destroy", NULL, NULL);
    gtk_signal_connect(GTK_OBJECT(clloButt), "clicked",
GTK_SIGNAL_FUNC(CollFunc), sock);
    gtk_signal_connect_object(GTK_OBJECT(exitButt), "clicked",
GTK_SI GNAL_FUNC(Destroy), GTK_OBJECT(mainWin));
    gtk_widget_show(mainWin);
    gtk_widget_show(exitButt);
    gtk_widget_show(clloButt);
    gtk_widget_show(clloLabl);
    gtk_widget_show(textbox);
    gtk_widget_show(table);
```

```
        gtk_main();
    }
```

上面的代码大概反映了 GTK+编程中常用的构件和函数。前面几章对 GTK+的编程进行了详细的讲解，这里就不再赘述。这里需要指出的是下面这个函数：

```
    gtk_signal_connect(GTK_OBJECT(clloButt), "clicked",
    GTK_SIGNAL_FUNC(CollFunc), sock);
```

这个函数主要是用来绑定 GTK+的信号与事件(前面也向读者详细介绍了)，也就是当用户按下了连接按钮后调用 CollFunc()这个函数。在客户端程序中，CollFunc()的主要作用是读取文本框中的字符，并且用这个来作为连接服务器的 IP，向服务器提出连接请求，连接成功就进入主界面，并且销毁连接界面的构件；连接不成功则返回到连接界面，用户需重新输入。CollFunc()函数如下(取自光盘的/src/chapter_16/clinet/conform.c 文件)：

```
    void CollFunc( GtkWidget *widget, gpointer data )
    {
        int *sock,result;
        char *text = NULL;
        sock = data;
        text = gtk_entry_get_text((GtkEntry *)textbox);
        *sock = make_clinet_sock(text,13000);
        if(*sock < 0)
    {
    return;
    }
        init_clinet(sock);
        mainform(sock);
        gtk_widget_destroy(mainWin);
    }
```

16.3.2 主界面

当用户端和服务器端连接成功后，就进入主窗口流程了，在主窗口建立前要接收一些从服务程序发送过来的数据，然后把这些数据存放到一些临时文件里面，等待具体创建窗口的时候调用这些数据来初始化界面。初始化主界面的流程如图 16.10 所示。

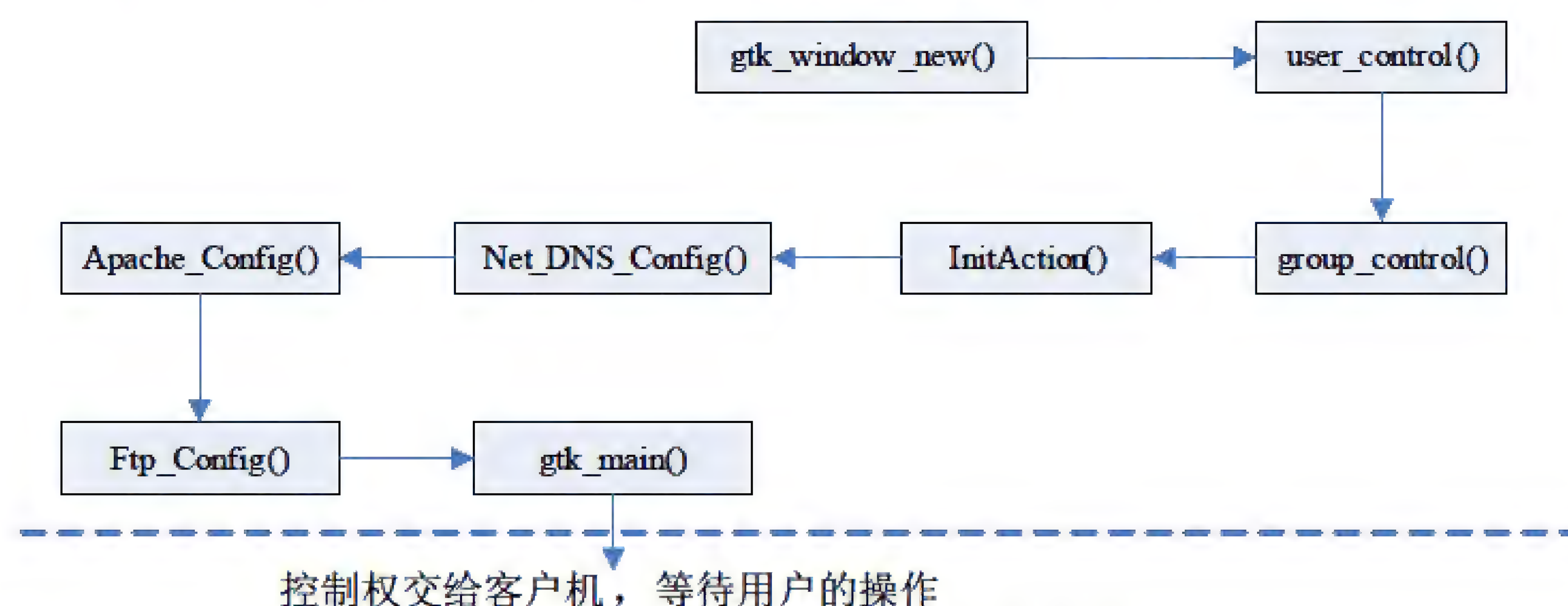


图 16.10 初始化主界面的流程图

主窗口以标签选项的方式包含了各个功能模块的操作页面，当用户单击不同的选项时将显示相应的配置界面，初始时默认的是用户管理操作的界面，如图 16.11 所示。

下面仅以用户管理操作的界面来向读者演示软件的使用。在用户管理操作界面中，我们主要实现的是对 Linux 系统用户的添加、删除和修改。

(1) 添加用户：当用户单击添加按钮“UserAdd”后，就调用了 User_Add()函数对这个事件来响应。其功能是创建一个临时的添加用户窗口，让用户输入要添加的用户的用户名、全名和密码，在判断了输入文本框都不为空之后，客户程序先传送给服务器一个“user”字符串，再传递一个“adduser”字符串，这些传送的数据是为了让服务器能够首先通过判断这些字符串进入到添加用户的流程。添加用户窗口的界面如图 16.12 所示。



图 16.11　用户操作主界面

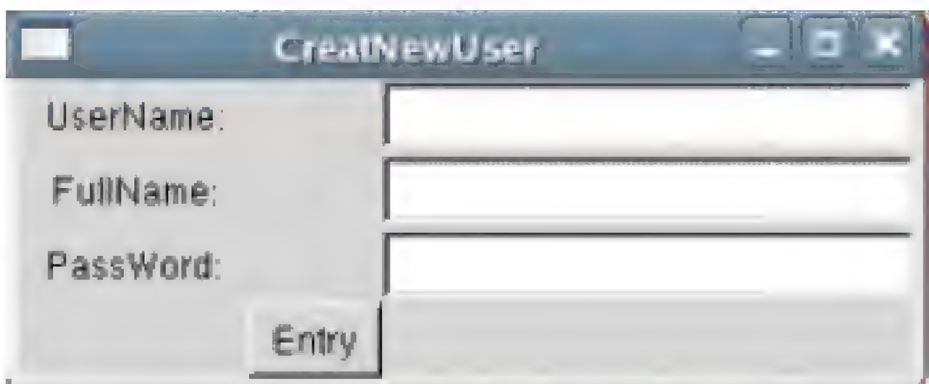
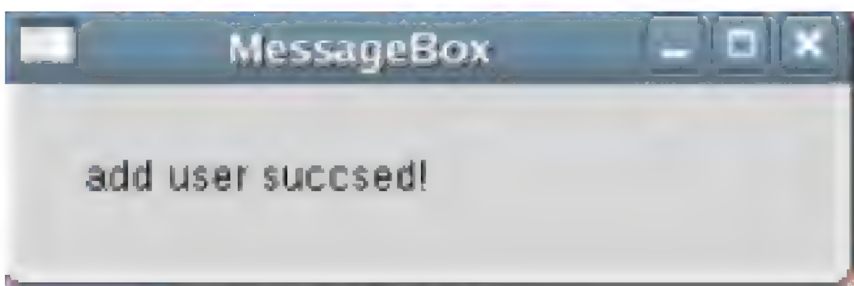
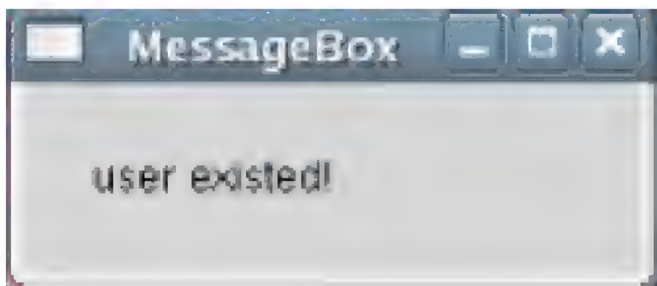


图 16.12　添加用户窗口

单击图 16.12 中的“Entry”按钮后，客户程序便从文本框中读取字符串，再判断字符串是否为空，若不为空就把 UserName、FullName、Password 所对应的字符串分别传递给服务器端。让服务器端通过传递过来的数据实现添加用户的操作，然后再从服务器端读取操作的结果，并用一个消息盒子“MessageBox”将返回信息显示给用户，如图 16.13 所示。图(a)是成功添加用户的返回信息，而当客户端输入的用户名在服务器中已存在时则返回图(b)的窗口。



(a)



(b)

图 16.13　反回信息“成功添加”和“用户已存在”

(2) 删除用户：当用户单击选中了列表中(列表中显示了当前服务器系统中存在的用户)的要删除的用户(比如 terrycheop 用户)，客户程序便调用 selection_call_back()函数。把用户所选中的列表框的用户的信息保存在全局变量中。当单击图 16.11 中的删除按钮“UserDel”后，便调用 User_Del()函数进行响应。User_Del()函数先向服务器发送一个“user”字符串，接着再发送一个“deluser”字符串，然后就把要删除的用户名发送给服务器程序。服务器操作完毕后，再将操作结果发送给客户端，客户端显示给用户。

(3) 修改用户：当用户从列表框里面选中了要修改的用户后，和删除用户操作一样会调用 selection_call_back()函数，把操作用户所选中的用户名(比如选中 cheops 用户)保存在一个全局变量中。当单击图 16.11 中的“Property”修改按钮后，客户端创建一个临时的窗口，用来显示当前所选中的用户属性，并向服务器程序发送一个“user”字符串，其界面如图 16.14 所示。

在这个界面中用户便可以输入想要修改的用户属性值。



图 16.14　修改用户窗口

在图 16.14 中，用户属性的信息填写/修改完毕后，单击“OK”按钮所触发的事件是先向服务器发送一个“property”字符串，然后读取文本框中的数据，并把这些字符串数据一起发送给服务器程序。单击“cancel”按钮所触发的事件是向服务器发送一个“exit”字符串。

(4) 刷新界面：图 16.11 中的刷新按钮“Refresh”所触发的事件是调用 UserScan()函数。这个函数主要作用是向服务器端发送一个“userscan”字符串，然后服务程序会读取当前系统下的用户管理配置文件/etc/passwd 中的内容，并且将读取的内容全部发送到客户程序，客户程序把服务程序传过来的数据存储在临时的文件里面，然后把列表框中的数据清空，再从临时文件中读取当前的用户信息，重新显示在列表框中。

鉴于篇幅有限，关于其他选项的操作就不在此一一列举了，有兴趣的读者可将本书附带光盘中的源代码复制至 Linux 主机下，进入相应的目录后进行试验。需要提醒读者的是，由于修改系统的配置文件具有用户权限的要求，所以试验本章所设计的软件时应该是以超级用户 root 登录系统的。另外，对于配置文件的修改具有一定的风险性，它可能导致系统的某个服务无法正常运行，所以读者应小心使用。笔者的建议是阅读光盘中的源代码，这样还会使你对 Linux 下的 C 编程有更加深入的认识。

16.4 本章小结

本章向读者讲述了一个基于 C/S 模式实现的类似于 Webmin 的 Linux 系统远程管理工具。该工具软件的主要功能是实现对 Linux 系统用户和组的添加、修改和删除；对系统中的应用服务(如：DNS、FTP、Apache、系统启动服务管理)进行管理和配置，这些服务的远程配置主要是通过修改相应的服务配置文本文件来实现的。通过本章的讲解，读者应该对 Linux 下的文件 I/O 操作、相关系统服务的管理有更深层次的了解和认识，以及进一步掌握 GTK+图形界面编程，套接字 Socket 网络编程的使用方法。

第17章

Linux下简易防火墙软件的设计

通过在本地网络和外部网络之间建立一道屏障，从而控制和管理进出网络的数据。网络管理控制系统的核心便是制定一套完整的网络控制指令集和设计控制管理的功能模块。本章将带领读者在 Linux-2.4.20-8 内核下完成简易防火墙软件的设计，使用控制管理命令实现对网络数据的管理。控制和管理模块的设计使用了 Netfilter 数据控制过滤机制来实现对网络数据报的管理。模块可以实现对固定端口、网页访问，以及不同数据协议类型的数据进行管理和控制。



本章内容：

◎ Netfilter 基础概述。
◎ 软件设计概述。
◎ 用 Netfilter 设计控制端功能模块。
◎ 软件功能测试。

17.1 Netfilter 基础

Netfilter 是 Linux 2.4 内核下实现 IP 数据包过滤、数据包处理、NAT(Network Address Translation，网络地址转换)等的功能框架，它是目前 Linux 2.4 内核中最流行的防火墙构建平台。本章所要设计的软件正是基于 Netfilter 的过滤机制对网络数据报进行管理和控制的。在介绍软件的设计之前，本节向读者介绍 Netfilter 的基础知识。

17.1.1 什么是 Netfilter

Netfilter 是新一代的 Linux 防火墙机制。Netfilter 采用模块化设计，具有良好的可扩充性，其重要工具模块 IPTables 连接到 Netfilter 的架构中，并允许使用者对数据报进行过滤、地址转换、处理等操作。Netfilter 提供了一个框架，将对网络代码的直接干涉降到最低，并允许用规定的接口将其他包处理代码以模块的形式添加到内核中，具有极强的灵活性。

Netfilter 是目前 Linux 2.4 内核中最流行的防火墙构建平台。通俗地讲，Netfilter 架构就是在整个网络流程的若干位置放置了一些检测点，称之为“钩子(HOOK)”，而在每个检测点上登记了一些处理函数来对 IP 数据包进行一些处理，比如包过滤、NAT 等，甚至可以是用户自定义的功能。

Netfilter 比以前任何一版 Linux 内核的网络管理控制子系统都要完善强大。Netfilter 提供了一个抽象、通用化的框架，该框架定义的一个子功能的实现就是包过滤控制子系统。Netfilter 框架包含以下 3 部分：

(1) 给网络协议(IPv4、IPv6 等)定义一套钩子函数，也可以称为 HOOK 函数(IPv4 中定义了 5 个钩子函数)，这些钩子函数在数据报流过协议栈的几个关键点被调用。在这几个点中，协议栈将把数据报及钩子函数标号作为参数调用 Netfilter 框架。

(2) 内核的任何模块可以对每种协议的一个或多个钩子进行注册，实现挂接。当某个数据包被传递给 Netfilter 框架时，内核能检测出是否有某个模块对该协议和相应的钩子函数进行了注册。若注册了，则调用该模块注册时使用的回调函数，这样这些模块就有机会检查(可能还会修改)该数据包、丢弃该数据包，以及指示 Netfilter 将该数据包传入用户空间的队列。

(3) 对于那些排队的数据包，将被传递给系统的用户空间异步地进行处理。一个用户进程能够检查数据包、修改数据包，甚至可以重新将该数据包通过离开内核时的同一个钩子函数再次注入内核中。

所有的网络数据包过滤、NAT 等处理都是基于 Netfilter 框架的，这使得内核的网络代码中不再有到处都是的、混乱的修改数据包的代码了。当前 Netfilter 框架已经在 IPv4、IPv6、Decnet、X.25 等网络协议栈中被实现。

提 示

读者可以查看 Linux 内核源代码中文件名以“netfilter_”开头的头文件。通常，这些头文件位于/usr/src/linux-2.4.20-8/include/linux 目录。“2.4.20-8”是笔者所使用的 Linux 主机系统版本。另外，Linux 内核源代码的存放目录默认为/usr/src/linux-2.4.20-8，下文将不再写出全部的绝对路径名。

另外，针对 IPv4，为方便读者查阅系统中的源代码，这里先给出本章所要讲述的 Netfilter 机制中将会涉及的文件：

- Netfilter 源文件：net/core/netfilter.c。
- Netfilter 头文件：include/linux/netfilter.h。
- IPv4 源文件：net/ipv4/netfilter/*.c。
- IPv4 头文件：include/linux/netfilter_ipv4.h、include/linux/netfilter_ipv4/*.h。
- IPv4 协议栈主体的部分 C 文件，特别是与数据报传送过程有关的部分：net/ipv4/ip_input.c、net/ipv4/ip_forward.c、net/ipv4/ip_output.c、net/ipv4/ip_fragment.c 等。

Netfilter 的具体功能模块包括数据报过滤模块(Filter)、连接跟踪模块(Conntrack)、网络地址转换模块(NAT)、数据报修改模块(Mangle)和其他高级功能模块。

Linux 内核模块可以对一个或多个钩子函数进行注册挂接，并且在数据报经过这些钩子函数时被调用，从而使模块可以修改这些数据报，并向 Netfilter 返回不同的值，它们的取值及其含义如表 17.1 所示。这些值定义于头文件 include/linux/netfilter.h 中。

表 17.1 HOOK 函数的返回值

返 回 值	含 义
NF_ACCEPT	继续正常传输数据报
NF_DROP	丢弃该数据报，不再传输
NF_STOLEN	模块接管该数据报，不要继续传输该数据报
NF_QUEUE	对该数据报进行排队(通常用于将数据报给用户空间的进程进行处理)
NF_REPEAT	再次调用该钩子函数

下面向读者介绍 Netfilter 框架的 HOOK 函数机制。考虑到 IPv4 仍是目前网络应用的主流，本章所要讲述的课题也是在 IPv4 网络协议下的设计与实现，故下面对于 Netfilter 的讲解将是特指在 IPv4 网络协议的环境下。

17.1.2 Netfilter 的 HOOK 机制

Netfilter 实际上是嵌入在 Linux 内核 IP 协议栈的一系列系统调用入口，设置在报文处理的路径上。网络报文按照来源和去向，可以分为流入的、流经的和流出的 3 类，其中流入和流经的报文需要经过路由才能区分，流经和流出的报文则需要经过投递，此外，流经的报文还有一个 FORWARD 的过程，即从一个网络适配器(Network Interface Card，简称 NIC，也叫作网卡)转到另一个网络适配器。Netfilter 就是根据网络报文的流向，在其中几个点插入处理过程，如

图 17.1 所示。

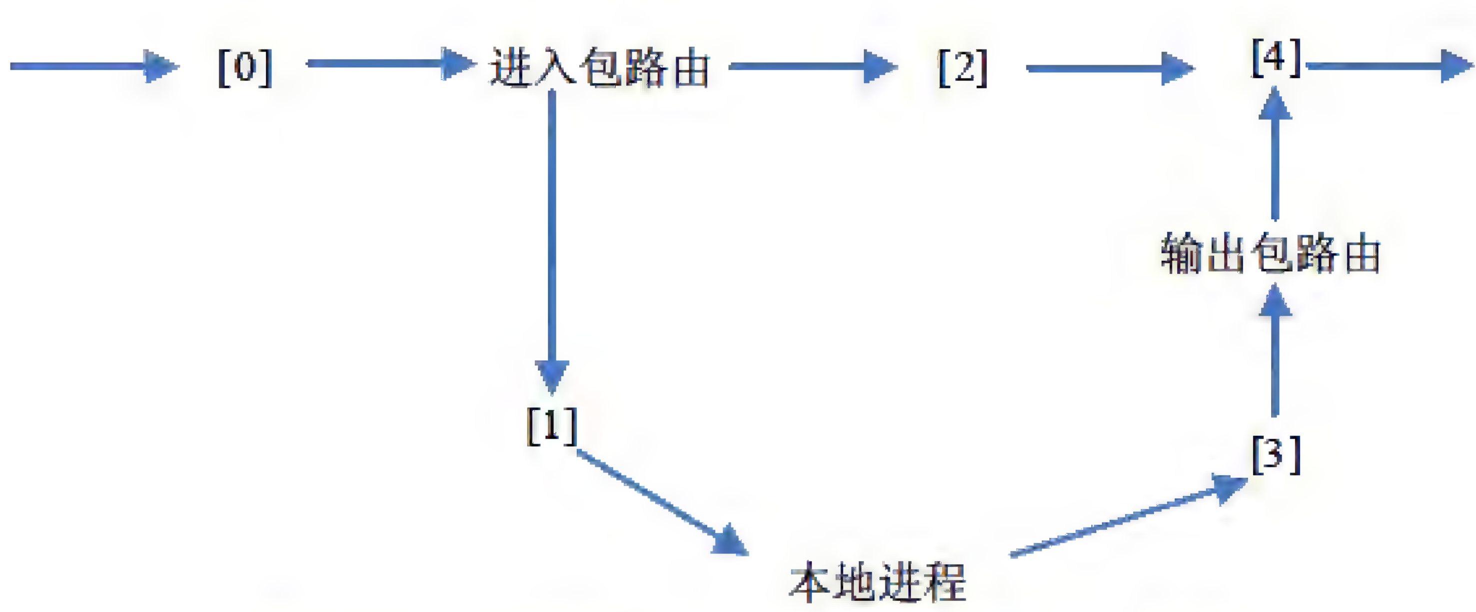


图 17.1　Netfilter 在 IPv4 中的框架示意图

从图 17.1 中可以看到，IPv4 协议下的 Netfilter 框架共有 5 个钩子函数。在文件 include/linux/netfilter_ipv4.h 中分别将它们定义为 5 个常量标识符(hooknum)，源代码定义形式如下：

```
/* IP Hooks */
#define NF_IP_PRE_ROUTING  0
#define NF_IP_LOCAL_IN 1
#define NF_IP_FORWARD2
#define NF_IP_LOCAL_OUT 3
#define NF_IP_POST_ROUTING  4
```

它们的含义分别如下([]中的数字为各自的 hooknum)：

- [0]NF_IP_PRE_ROUTING：在报文做路由以前执行。
- [1]NF_IP_LOCAL_IN：在流入本地的报文做路由以后执行。
- [2]NF_IP_FORWARD：在报文转向另一个 NIC 以前执行。
- [3]NF_IP_LOCAL_OUT：在本地报文做流出路由之前执行。
- [4]NF_IP_POST_ROUTING：在报文流出以前执行。

简单地说，数据报经过各个 HOOK 的流程可描述如下：

当数据报经过了完整性检查后，数据报就从左边进入系统，进行 IP 校验以后，数据报经过第一个钩子函数 NF_IP_PRE_ROUTING 进行处理，然后就进入路由代码，其决定了该数据报是需要转发还是发给本机。若该数据报是发给本机，则该数据报经过第二个钩子函数 NF_IP_LOCAL_IN 处理以后传递给上层协议(应用层协议)。若该数据报应该被转发，则它被钩子函数 NF_IP_FORWARD 处理，转发后的数据报经过最后一个钩子函数 NF_IP_POST_ROUTING 的处理以后，再传输到网络上。

本地产生的数据经过钩子函数 NF_IP_LOCAL_OUT 处理以后，进行路由选择处理，然后经过 NF_IP_POST_ROUTING 处理以后发送到网络上。

由此可见，5 个 HOOK 的位置，掌管了全部数据包的可能出入口，我们只要在对应的位置对数据报进行操作，就能实现对数据报的各种处理。

详细地说，各个 HOOK 及其在 IP 数据报传递过程中的具体位置，以及内核中相关的处理函数如图 17.2 所示。

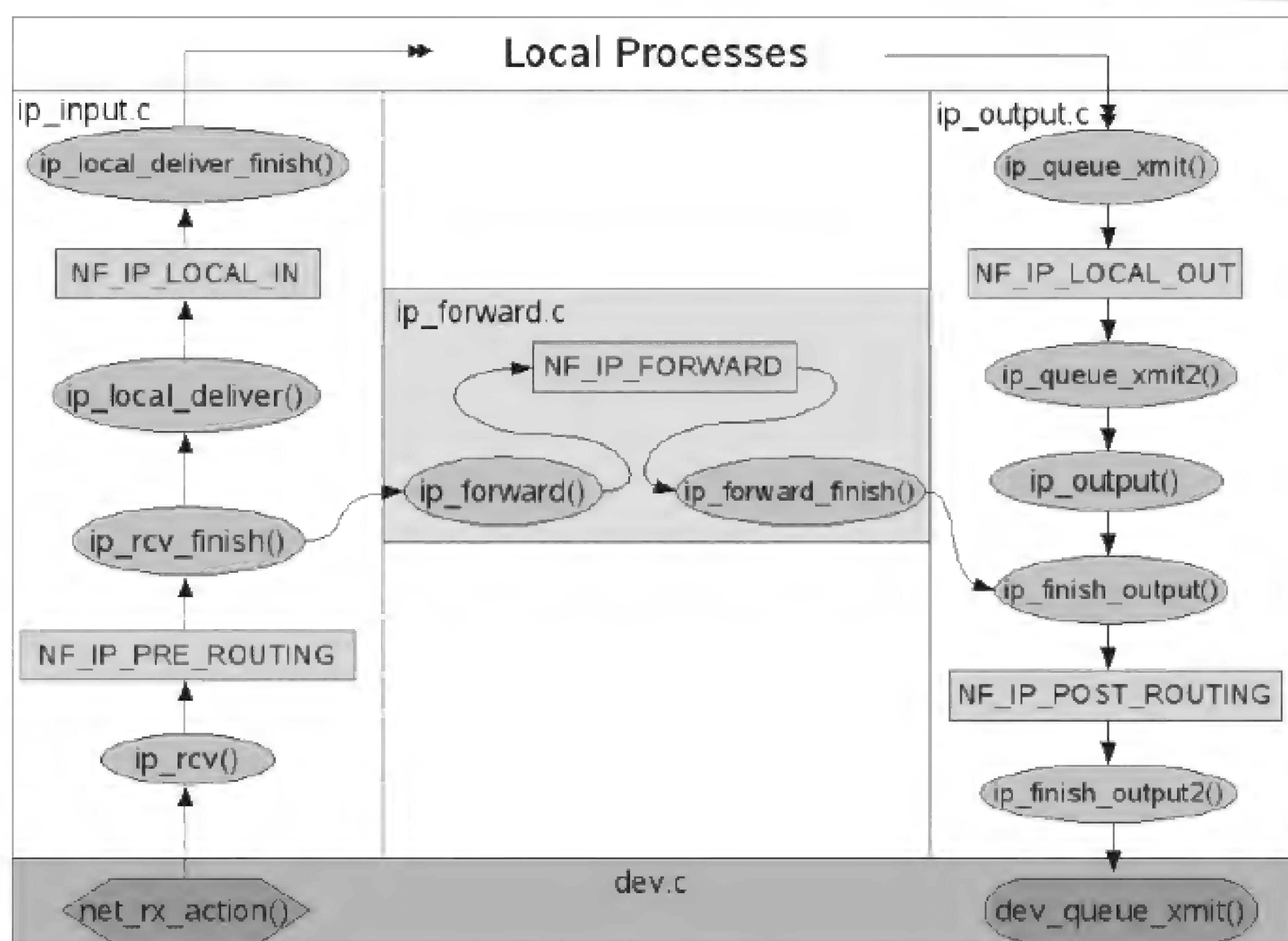


图 17.2　Netfilter-IPv4 的数据报过滤流程详图

对于图 17.2 所示流程，我们进行如下的解释。

(1) NF_IP_PRE_ROUTING

数据报在进入路由代码被处理之前，在 IP 数据报接收函数 ip_rcv()(位于 net/ipv4/ip_input.c，Line379)的最后，也就是在传入的数据报被处理之前经过这个 HOOK。在 ip_rcv()中挂接这个 HOOK 之前，进行的是一些与数据报文类型、长度、版本有关的检查。经过这个 HOOK 处理之后，数据报进入 ip_rcv_finish()(位于 net/ipv4/ip_input.c，Line306)，进行查询路由表的工作，并判断该数据报是发给本地机器还是进行转发。在这个 HOOK 上主要是对数据报进行报头检测处理，以捕获异常情况。

(2) NF_IP_LOCAL_IN

目的地为本地主机的数据报在 IP 数据报本地投递函数 ip_local_deliver()(位于 net/ipv4/ip_input.c，Line290)的最后经过这个 HOOK。经过这个 HOOK 处理之后，数据报进入 ip_local_deliver_finish()(位于 net/ipv4/ip_input.c，Line219)。这样，IPTables 模块就可以利用这个 HOOK 对应的 INPUT 规则链表来对数据报进行规则匹配的筛选了。防火墙一般建立在这个 HOOK 上。

(3) NF_IP_FORWARD

目的地非本地主机的数据报，包括被 NAT 修改过地址的数据报，都要在 IP 数据报转发函数 ip_forward()(位于 net/ipv4/ip_forward.c，Line73)的最后经过这个 HOOK。经过这个 HOOK 处理之后，数据报进入 ip_forward_finish()(位于 net/ipv4/ip_forward.c，Line44)。另外，在 net/ipv4/ipmr.c 中的 ipmr_queue_xmit()函数(Line1119)，最后也会经过这个 HOOK(ipmr 为多播相关，是在需要通过路由转发多播数据时的处理)。这样，IPTables 模块就可以利用这个 HOOK 对应的 FORWARD 规则链表来对数据报进行规则匹配的筛选了。

(4) NF_IP_LOCAL_OUT

本地主机发出的数据报在 IP 数据报构建/发送函数 ip_queue_xmit()(位于 net/ipv4/ip_output.c，Line339)及 ip_build_and_send_pkt()(位于 net/ipv4/ip_output.c，Line122)的最后经过这个 HOOK(在数据报处理中，前者最为常用，后者用于那些不传输有效数据的 SYN/ACK 确认包)。

经过这个 HOOK 的处理后，数据报进入 ip_queue_xmit2()(位于 net/ipv4/ip_output.c，Line281)。另外，在 ip_build_xmit_slow()(位于 net/ipv4/ip_output.c，Line429)和 ip_build_xmit()(位于 net/ipv4/ip_output.c，Line638)中用于进行错误检测；在 igmp_send_report()(位于 net/ipv4/igmp.c，Line195)的最后也经过了这个 HOOK，进行多播时相关的处理。这样，IPTables 模块就可以利用这个 HOOK 对应的 OUTPUT 规则链表来对数据报进行规则匹配的筛选了。

(5) NF_IP_POST_ROUTING

所有的数据报，包括源地址为本地主机和非本地主机的，在通过网络设备离开本地主机之前，在 IP 数据报发送函数 ip_finish_output()(位于 net/ipv4/ip_output.c，Line184)的最后经过这个 HOOK。经过这个 HOOK 处理后，数据报进入 ip_finish_output2()(位于 net/ipv4/ip_output.c，Line160)。另外，在函数 ip_mc_output()(位于 net/ipv4/ip_output.c，Line195)中在克隆新的网络缓存 skb 时，也经过了这个 HOOK 的处理。

另外，入口为 net_rx_action()(位于 net/core/dev.c，Line1602)，作用是将数据报一个个地从 CPU 的输入队列中拿出，然后传递给协议处理例程。出口为 dev_queue_xmit()(位于 net/core/dev.c，Line1035)，这个函数被高层协议的实例使用，以数据结构 struct sk_buff *skb 的形式在网络设备上发送数据报。

17.1.3 HOOK 的调用

IPv4 协议栈为了实现对 Netfilter 架构的支持，在数据报经过 IPv4 协议栈的过程中，仔细选择了 5 个参考点(检测点)：NF_IP_PRE_ROUTING、NF_IP_LOCAL_IN、NF_IP_FORWARD、NF_IP_LOCAL_OUT 和 NF_IP_POST_ROUTING，分别对应 IP 层 5 个不同的位置。在这 5 个参考点上，各引入了对 NF_HOOK()宏函数的一个相应的调用。

在 Linux 内核中，netfilter.h 文件定义了 NF_HOOK()宏函数。对于 NF_HOOK 宏的定义提炼如下(取自 include/linux/netfilter.h 文件)：

```
#ifdef CONFIG_NETFILTER
#define NF_HOOK(pf,hook,skb,indev,outdev,okfn)
(list_empty(&nf_hooks[(pf)][(hook)]) ?(okfn)(skb):
nf_hook_slow((pf),(hook),(skb),(indev),(outdev),(okfn))
#else       /* !CONFIG_NETFILTER */
#define NF_HOOK(pf,hook,skb,indev,outdev,okfn) (okfn)(skb)
#endif      /* CONFIG_NETFILTER */
```

从中可以看到，Linux 内核先调用 list_empty()函数检查 HOOK 点的存储数组 nf_hooks 是否为空，为空则表示没有 HOOK 注册，则直接调用 okfn 继续处理。如果不为空，则转入执行 nf_hook_slow()函数。这样，用户在编译 Linux 内核时便可以通过定义 CONFIG_NETFILTER 与否来决定是否把 Netfilter 代码编译进内核。

nf_hook_slow()函数(位于 net/core/netfilter.c，Line449)的主要工作是读取 nf_hooks 二维数组的数据，遍历所有的 nf_hook_ops 结构，并调用 nf_hookfn()处理各个数据报。

从函数的名称来看，也可以把 IPv4 协议栈上的这 5 个参考点，形象地看成是 5 个钩子。当数据报在 IPv4 协议栈上途经这 5 个钩子时，就会被 Netfilter 模块“钓”上来进行处理，并根据返回值来决定数据报的下一步“命运”，比如被继续传输或者被丢弃。HOOK 的调用过程如

图 17.3 所示。

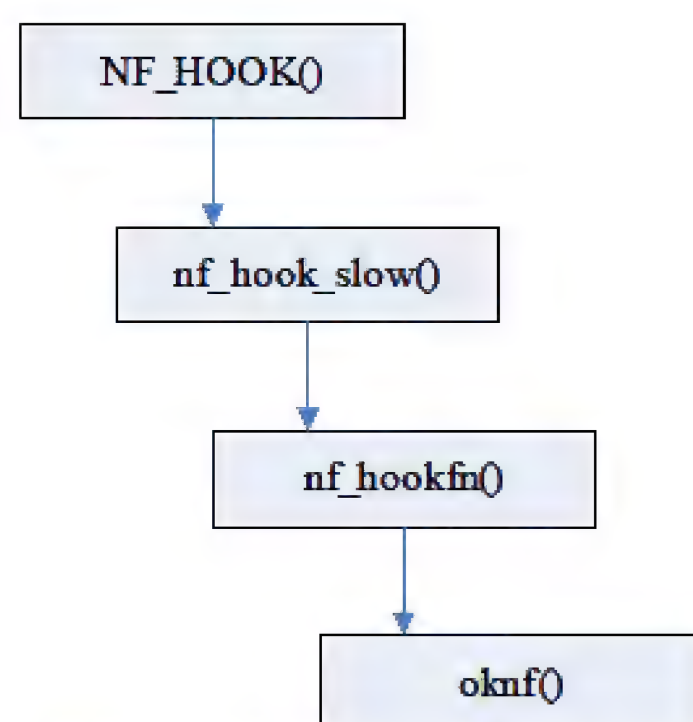


图 17.3　HOOK 的调用过程

下面简要说明 NF_HOOK()宏函数中的各个参数的含义：

- pf：协议族标识，相关的有效协议族列表位于 include/linux/socket.h 文件。对于 IPv4，应该使用协议族 PF_INET(或 AF_INET)。
- hook：HOOK 标识，即前文中所说的 5 个 HOOK 对应的 hooknum。
- skb：是含有需要被处理包的 sk_buuff 数据结构的指针。sk_buff 是 Linux 的网络缓存，指那些 Linux 内核处理 IP 分组报文的缓存，即套接字缓冲区。网卡收到 IP 分组报文后，将它们放入 sk_buff，然后再传送给网络堆栈。网络堆栈几乎一直要用到 sk_buff，其定义在 include/linux/skbuff.h 头文件中。
- indev：输入设备，收到数据报的网络设备的 net_device 数据结构指针，即数据报到达的接口。用于 NF_IP_PRE_ROUTING 和 NF_IP_LOCAL_IN 两个 HOOK 函数。
- outdev：输出设备，数据报离开本地时所要使用的网络设备的 net_device 数据结构指针。用于 NF_IP_LOCAL_OUT 和 NF_IP_POST_ROUTING 两个 HOOK 函数。需要注意的是，在通常情况下，在一次 HOOK 调用中，indev 和 outdev 只有一个参数会被使用。
- okfn：下一步将要处理的函数。即如果有 HOOK 函数，则处理完所有的 HOOK 函数，且所有向该 HOOK 注册过的筛选函数都返回 NF_ACCEPT 时(参考表 17.1)，调用这个函数继续处理；如果没有注册任何 HOOK，则直接调用此函数。它的 5 个参数将由宏 NF_HOOK 传入。

17.1.4　HOOK 的实现

上面已向读者提到，在 Netfilter 机制中，如果 Netfilter 的钩子函数被调用，数据包就进入 nf_hook_slow()函数的处理，此函数主要是根据二维数组 nf_hooks 开始处理数据包。更准确一点来说，上文所说的 IPv4 协议栈上的 5 个参考点，并不是防火墙的钩子函数，而是在此点内核允许放置防火墙的钩子函数。

在每一个参考点中，都可以让 Netfilter 放置一个或多个处理函数，来处理经过参考点的数据包，而 Netfilter 的钩子函数则放在了 nf_hooks 数组里面。这些钩子函数用结构体 struct nf_hook_ops 给予描述(位于 include/linux/netfilter.h，Line 44)，其声明如下：

```
struct nf_hook_ops
{
struct list_head list;
```

```
    nf_hookfn * hook;
    /*指向处理函数的指针，其 5 个参数分别对应于 NF_HOOK 中的第 2~6 个参数*/
    int pf;
    int hooknum;
    int priority;        /*优先级，值越小，优先级越高*/
};
```

在使用 Netfilter 实现管理功能的模块初始过程中，它会调用 nf_register_hook()向 Netfilter 的核心代码注册钩子函数。在这个注册的过程中，也就是将钩子函数放在参考点的过程。钩子函数的具体位置，由 nf_hooks 数组的下标具体说明。

另外，对于 Netfilter 所提供的框架，这些钩子函数都将会调用 mymodules()函数。该函数执行完后，将返回通用的防火墙策略之一，如 NF_ACCEPT 等。该函数原型如下：

```
unsigned int mymodules(struct sk_buff **pskb, unsigned int hook, const struct net_device * in,
const struct net_device * out, struct ipt_table * table, void * userdata);
```

在函数的参数列表中，pskb 指向传入的待处理的网络协议包，hook 代表在哪个钩子点处理该数据包，in 表示网络包传入的网络设备名，out 表示网络包传出的网络设备名，table 指向用户自定义的规则表，userdata 是指向用户数据的指针。

对于用户自定义的规则表，经过一些处理后送到系统内核空间，内核空间将用一些数据结构进行标识。

在 Netfilter 框架中，一条规则分为 3 个部分，分别由 3 个数据结构来代表：

- struct ipt_entry：主要用来匹配 IP 头。
- struct ip_match：额外的匹配(TCP 头、MAC 地址等)。
- struct ip_target：除默认的动作外(如 NF_ACCEPT、NF_DROP)，还可以增加新的动作(如 NF_REJECT)。

mymodules()函数就是按照规则表中存储的一条又一条的规则来处理数据包。但并不是所有的规则都一一来匹配数据包，数据包只与相应的参考点的规则相匹配。这个机制，就为每个规则表实现了多个规则链，而每个规则链上又有多个规则。

说 明

匹配(match)是 iptables 命令的可选部分，它用于匹配数据包的源地址/源端口、目的地址/目的端口、协议等。匹配分为两大类：通用匹配和特定于协议的匹配。目标(target)是由规则指定的操作，对与那些规则匹配的数据包则执行这些操作。除了默认的目标之外，还增加新的目标选项。

最后，在使用 HOOK 之前需要对其进行注册，而使用完毕之后则需要对其注销。HOOK 的注册与注销分别是通过 nf_register_hook()函数和 nf_unregister_hook()函数(分别位于 net/core/netfilter.c，Line60，76)实现的，其参数均为一个 nf_hook_ops 结构，两者的实现也非常简单。

nf_register_hook()的工作是首先遍历二维数组 nf_hools[][]，由 HOOK 的优先级确定在 HOOK 链表中的位置，然后根据优先级将该 HOOK 的 nf_hook_ops 结构体加入链表。

nf_unregister_hook()的工作更加简单，其实就是将该 HOOK 的 nf_hook_ops 从链表中删除。

17.1.5 IPTables 简介

前文中已多次提到过 IPTables，它是 Netfilter 的一个重要工具模块，是 Linux 系统中经常使用的网络管理工具。

管理工具 IPTables 是 Linux 2.4 内核用来安装、维护、检查数据包规则的管理程序。在 Linux 2.4 网络管理控制体系结构中，数据处理的规则可以分为 4 类：IP 输入链(IP Input Chain)、IP 输出链(IP Output Chain)、IP 转发链(IP Forward Chain)、用户自定义链(User Defined Chain)。网络数据控制管理的规则指定包的格式和目标。当一个包进来时，内核使用输入链来决定数据包的命运。数据包沿着输入链一条规则一条规则地进行匹配，如果它通过了输入链的检查，内核将决定包下一步该发往何处(这一步叫路由)。假如该数据包是送往另一台机器的，内核就将调用转发链。数据包再沿着转发链一条规则一条规则地检查，如果不匹配，就进入目标值所指定的下一条链。这条规则链有可能是用户自己定义的规则链，或者是一个特定值：接受(ACCEPT)、否定(DENY)、拒绝(REJECT)、伪装(MASQ)、重定向(REDIRECT)或返回(RETURN)。

一个基本的 iptables 命令包含如下 5 个部分：

- 希望工作在哪个表上。
- 希望使用该表的哪个链。
- 进行何种操作，比如插入、添加、删除、修改等。
- 对特定规则的目标动作。
- 匹配数据报条件。

iptables 命令的基本语法如下：

```
iptables -t table –O peration chain -j target match(es)
```

选项“-O”指明 iptables 命令的操作动作，它可能的取值及含义有：“-A”在链尾添加一条规则；“-I”插入一条规则；“-D”删除一条规则；“-R”替代一条规则；“-L”列出所有规则。

选项“target”指明了 iptables 的基本目标动作(适用于所有的链)，它可能的取值及含义有：“ACCEPT”接收该数据报；“DROP”丢弃该数据报；“QUEUE”排队该数据报到用户空间；“RETURN”返回到前面调用的链；“Foobar”适用于用户自定义的链。

选项“match(es)”表示 iptables 的一些基本匹配条件(适用于所有的链)，它可能的取值及含义有：“-p”指定协议(tcp/udp/icmp/…)；“-s”源地址(ip address/masklen)；“-d”目的地址(ip address/masllen)；“-I”数据报输入接口；“-o ”数据报输出接口。

例如，如果希望添加一个规则，允许所有从任何地方到本地 SMTP 端口的连接，可以使用如下形式的命令：

```
iptables -t filter -A INPUT -j ACCEPT -p tcp --dport smtp
```

还有其他对规则进行操作的命令，如：清空链表、设置链默认策略、添加一个用户自定义的链等。

说　明

除了基本的操作，匹配和目标还具有各种扩展。这里只对 iptables 进行简单的讨论，关于 iptables 的详细使用，可以参考 man iptables 的手册，也可以参考 netfilter 的核心开发者 Paul Russell 编写的 Packet Filtering HOW-TO 和 NAT HOW-TO。

17.1.6　Netfilter 可以实现的控制功能

基于 HOOK 函数的机制，Netfilter 对网络数据报可以实现的基本控制功能包括包过滤、NAT、包处理、连线跟踪等。

1. 包过滤

包过滤的控制模块不会对数据包进行修改，只对数据包进行过滤。它通过钩子函数 NF_IP_LOCAL_IN、NF_IP_FORWARD 及 NF_IP_LOCAL_OUT 接入 Netfilter 框架。对于任何一个数据报只有一个地方对其进行过滤。Iptables 相对 ipchains 是一个巨大的改进，因为在 ipchains 中一个被转发的数据报会遍历 3 条链。Netfilter 的包过滤框架示意图如图 17.4 所示。

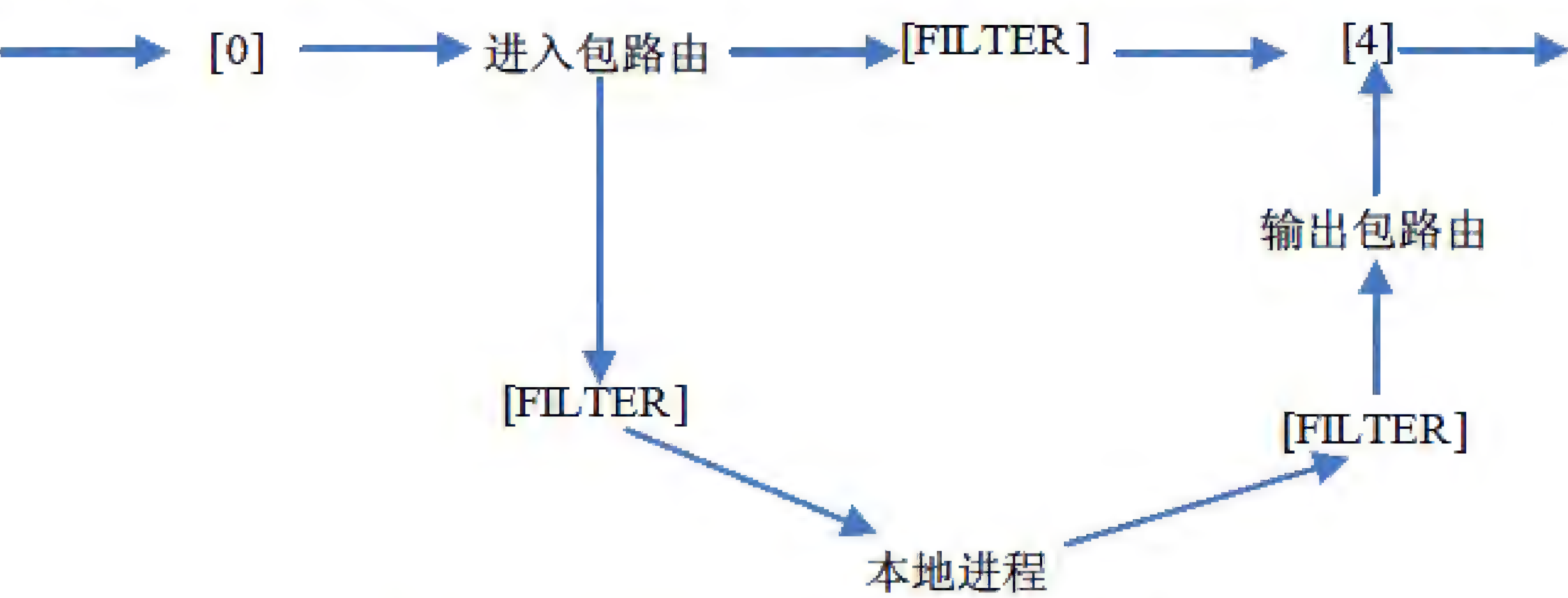


图 17.4　Netfilter 包过滤框架示意图

2. NAT

网络地址转换通过 NF_IP_PRE_ROUTING、NF_IP_POST_ROUTING 及 NF_IP_LOCAL_OUT 3 个钩子函数接入 Netfilter 框架。网络地址转换只对新连接的第一个数据包进行转换，随后的数据包将根据第一个数据包的结果进行同样的转换处理，它分为源地址转换和目的地址转换。

NF_IP_PRE_ROUTING 实现对转发的数据包的源地址进行地址转换，NF_IP_POST_ROUTING 对转发的数据包的目的地址进行地址转换，对于本地数据报的目的地址的转换则由 NF_IP_LOCAL_OUT 来实现。Netfilter 的地址转换框架示意图如图 17.5 所示。

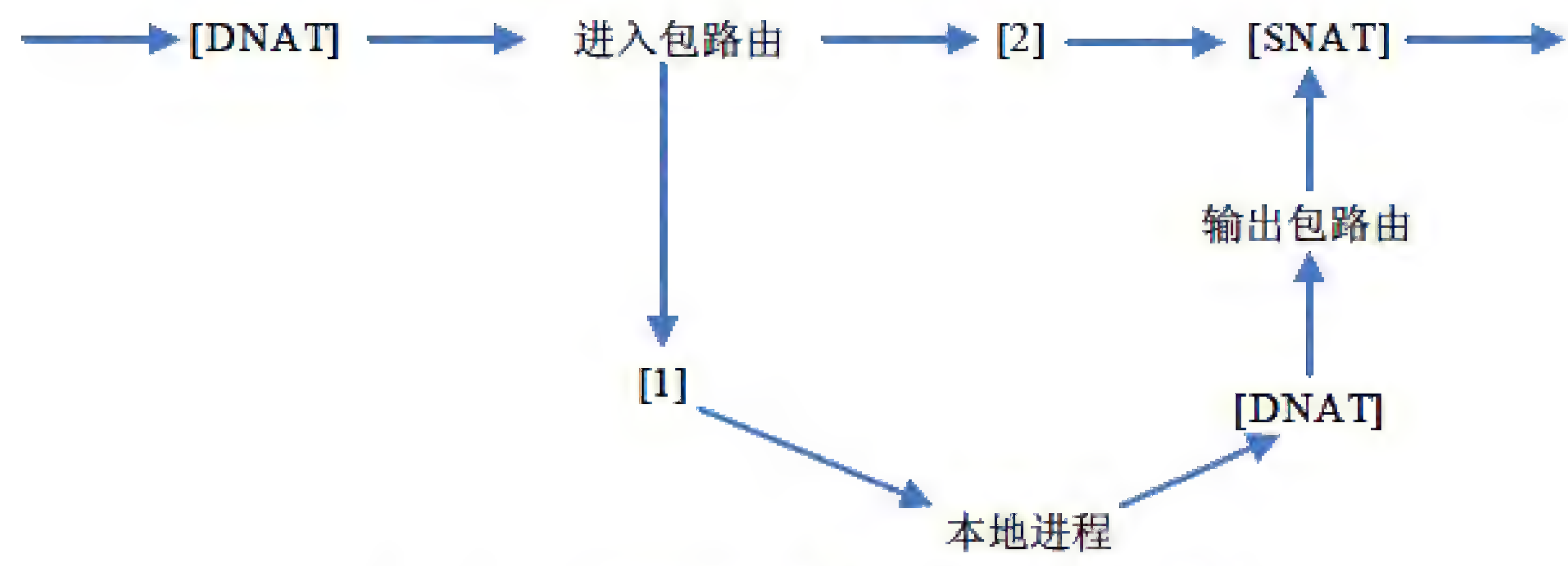


图 17.5　Netfilter 地址转换框架示意图

3. 数据包处理

数据包处理通过钩子函数 NF_IP_PRE_ROUTING 和 NF_IP_LOCAL_OUT 接入 Netfilter 框架。包处理可以实现对数据报的修改或给数据报附上一些外带数据。将框架中的钩子点 0 和钩子点 3 进行处理，即在下图中的 MANGLE 处进行处理。Netfilter 的数据包处理框架示意图如图 17.6 所示。

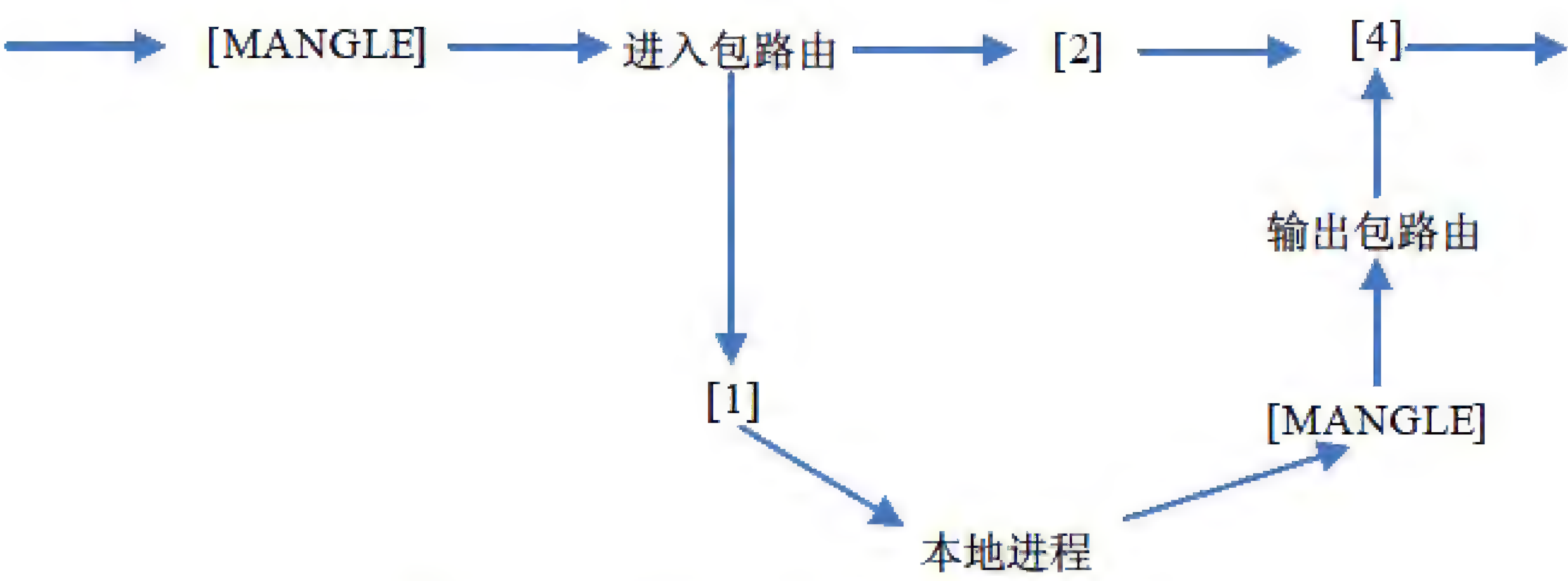


图 17.6　Netfilter 包处理框架示意图

4. 连线跟踪

连线跟踪(Connection Tracking)是包过滤、地址转换的基础，但其又作为一个独立的模块在运行，有了连线跟踪，动态包过滤及地址转换才能得以实现。

连线跟踪的工作原理是：检测第一个有效连接的状态，并根据这些信息决定网络数据包是否能够通过 Netfilter 功能模块。

连线跟踪在 Netfilter 框架中的 NF_IP_PRE_ROUTING、NF_IP_LOCAL_IN、NF_IP_POST_ROUTING、NF_IP_LOCAL_OUT 4 个地方被采用，连线跟踪的框架示意图如图 17.7 所示。

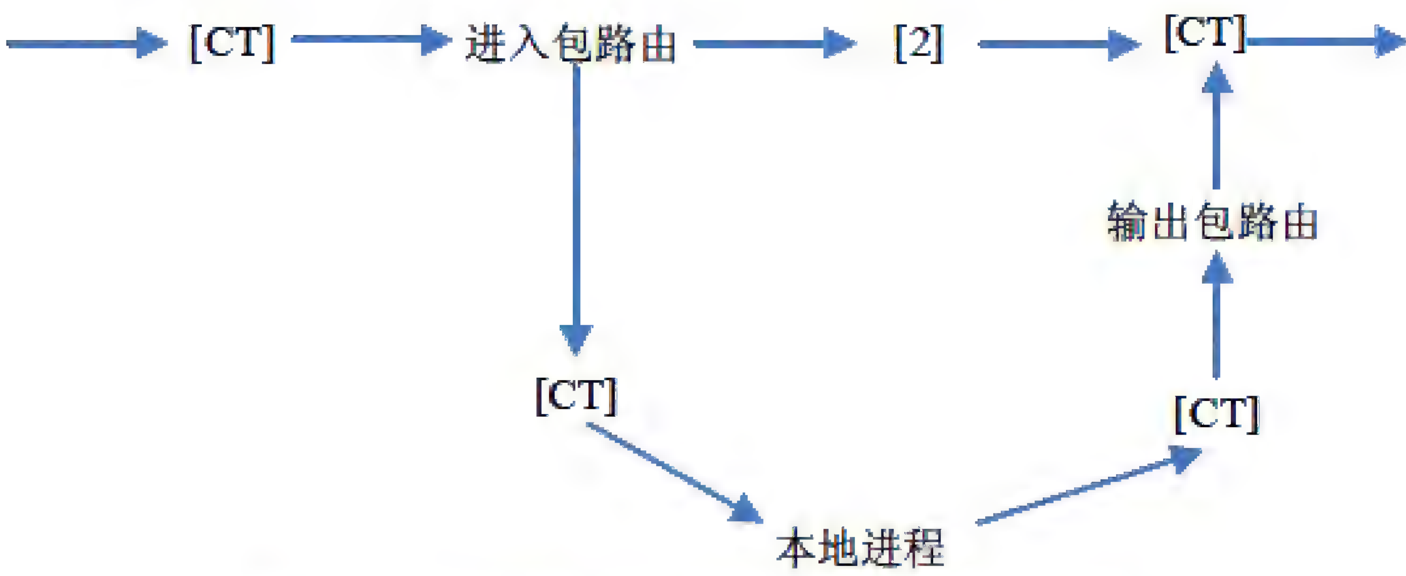


图 17.7　Netfilter 连线跟踪框架示意图

17.2 软件设计概述

在了解完 Netfilter 机制的基础之上，本节开始讲解 Linux 下简易防火墙软件的设计。该软件主要分为管理端和控制端，管理端接收用户的输入命令，控制端根据这些命令完成相应的控制操作，包括对 ICMP 网络数据，HTTP 站点，FTP 站点的管理控制。

17.2.1 软件整体框架

本章所要设计的 Linux 简易防火墙软件的控制管理框架主要分为两个部分：管理部分(管理

端)和控制部分(控制端)。不同于前面几章的设计实例，本章的实例并不需要图形用户界面，用基于字符的方式接受用户的输入命令，使课题的设计难度大大降低，也简化了代码的实现。而本章设计的重点在于使用 Linux 内核的 Netfilter 机制来编译程序的代码。

首先，软件的管理部分拥有一套完整的控制指令，管理端可以接收用户终端输入的字符，并负责将用户输入的字符命令发送给控制端(通过 TCP 协议通信)。控制端接收到管理端发送来的命令，便执行相应的操作。这些操作主要是通过管理端程序发出的控制指令对相应的应用程序进行动态的插入和卸载，使这些模块可以动态地加入 Linux 的内核并运行，或是从内核中卸载，恢复内核原来的功能。

在本章的课题中，控制端的功能实现了对一些比较常用的网络服务协议的控制。控制端主要分为 3 个模块：ICMP 管理控制模块、FTP 管理控制模块和 HTTP 管理控制模块。分别实现了对网络上 ICMP 数据报的过滤、收发，FTP 服务器资源的禁用、启用，以及 HTTP 网页访问禁用、启用等功能，其实质都是对网络数据报的检测、过滤及屏蔽作用，基本实现了对进出网络的数据的控制和管理功能，即实现了简易防火墙的功能(尽管操作系统实际使用的防火墙软件的功能要比这强大得多)。

Linux 下简易防火墙软件的管理控制系统的结构如图 17.8 所示。

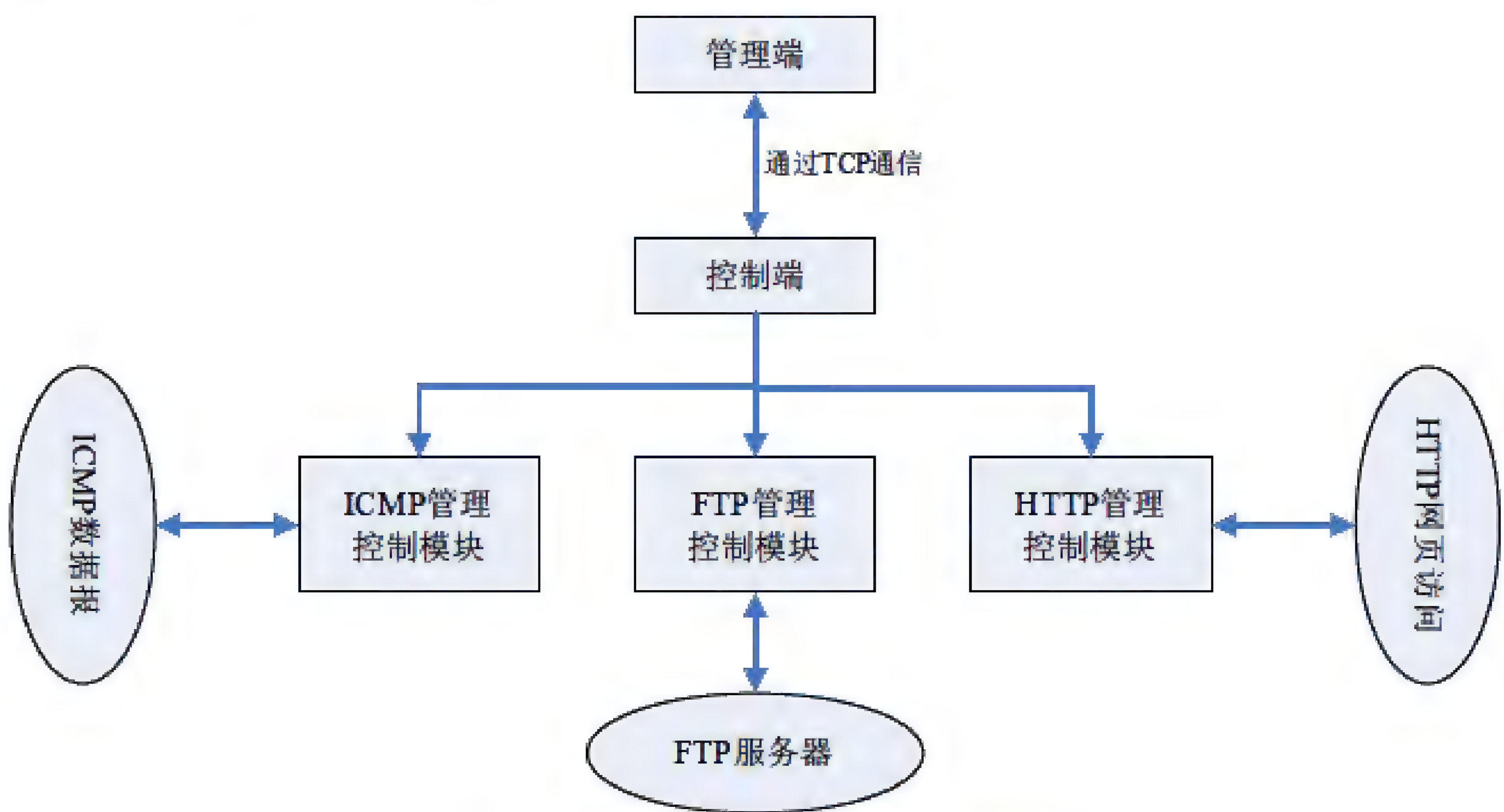


图 17.8　网络管理系统结构图

另外，在管理端与控制端的通信方面，程序代码中的网络套接字均是采用 TCP 4114 端口，并且为了方便对设计效果的检验，控制端的 IP 地址设为本机 IP，即 INADDR_ANY(参考 11.3.1 小节)。

在本章设计的软件中，Socket 通信的具体流程可简要描述如下：

(1) 控制端处于监听状态，等待管理端的连接请求。

(2) 管理端输入控制端的 IP 地址，向控制端请求建立连接。若连接成功，管理端会接收到控制端数组 buff 中的内容，即“hello”字符串。

(3) 成功建立 TCP 连接后，管理端将用户输入的控制字符存放在 sebuff 数组中，通过 send() 函数发送出去。

(4) 控制端通过 recv()函数接受管理端发来的控制字符，并且把字符存放进 resbuff 数组中。

(5) 控制端根据控制字符去执行相应的具体操作。

(6) 通信完毕使用 close()函数关闭套接字。

17.2.2 管理端的设计

管理端的主要功能是建立与控制端的 TCP 连接，以字符命令行的方式接收用户从终端的输入，并将字符命令发送给控制端。下面给出了主要部分代码及注释，全部源代码可参见光盘中的内容。

【程序 17.1】管理端核心代码及注释(取自光盘的/src/chapter_17/ke.c 文件)：

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

int main(int argc,char **argv)
{
int i=0;
char w;
char sebuff[5];
char recvbuff[100];
int sockfd;
struct sockaddr_in serveraddr;
if(argc!=2)
{
printf("usage:echo ip");
exit(0);
}
/*下面是创建 TCP 套接字，端口号为 4114*/
sockfd=socket(AF_INET,SOCK_STREAM,0);
bzero(&serveraddr,sizeof(serveraddr));
serveraddr.sin_family=AF_INET;
serveraddr.sin_port=htons(4114);
inet_pton(AF_INET,argv[1],&serveraddr.sin_addr);
connect(sockfd,(struct sockaddr*)&serveraddr,sizeof(serveraddr));
recv(sockfd,recvbuff,sizeof(recvbuff),0);/*用 recv 接受控制端数组 buff 中的数据*/
/*下面是打印管理端的字符命令信息，提示用户输入正确的字符命令*/
printf("%s\n",recvbuff);
printf("please input the message:\n");
printf("please input the 'a' to add the module.\n");
printf("please input the 'c' close the WEB servers.\n");
printf("please input the 'o' open the WEB servers.\n");
printf("please input the 'f' filter the ICMP packet.\n");
printf("please input the 'i' accept the ICMP packet.\n");
printf("please input the 't' close the FTP servers.\n");
printf("please input the 'p' start the FTP servers.\n");
printf("please input the 'r' rmmod the all modules.\n");
scanf("%c",&w);

while(w!='q')
```

```
{
sebuff[0]=w;
send(sockfd,sebuff,1,0);        /*发送字符命令到控制端*/
scanf("%c",&w);
switch(w)
{
case 'a':
printf("start the module.\n"); break;
case 'b':
printf("pause the connect.\n"); break;
case 'c':
printf("start close the WEB server.\n"); break;
case 'o':
printf("start open the WEB server.\n"); break;
case 'f':
printf("start filter the ICMP packet.\n"); break;
case 'i':
printf("start receive the ICMP packet.\n"); break;
case 't':
printf("start close the FTP servers.\n"); break;
case 'p':
printf("start open the FTP servers.\n"); break;
case 'z':
printf("test.\n"); break;
case'r':
printf("rmmod the control modules.\n"); break;
default:
printf("-----------------------.\n");
}
}
…………/*其余部分省略*/
close(sockfd);        /*通信完毕，关闭套接字*/
}
```

17.2.3　控制端的设计

控制端为一个后台执行程序，控制端在和管理端建立 TCP 连接后，接收来自管理端的字符命令，然后调用相应的功能模块(有 3 个模块)，执行具体的网络数据报的控制操作。下面给出了控制端的主要部分代码及注释，全部源代码可参见光盘中的内容。

【程序 17.2】控制端核心代码及注释(取自光盘的/src/chapter_17/fu.c 文件)：

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <unistd.h>
```

```
int main(int argc,char *argv[])
{
char temp;
int a;
char resbuff[5];     /*定义接收缓冲区数组*/
int listensock,connsock;
struct sockaddr_in serveraddr;
const char buff[]="hello\r\n";   /*连接建立成功时，发送给管理端*/
/*下面是建立控制端的套接字*/
listensock=socket(AF_INET,SOCK_STREAM,0);
bzero(&serveraddr,sizeof(serveraddr));
serveraddr.sin_family=AF_INET;
serveraddr.sin_addr.s_addr=htonl(INADDR_ANY);
serveraddr.sin_port=htons(4114);
bind(listensock,(struct sockaddr*)&serveraddr,sizeof(serveraddr));
listen(listensock,4);
connsock=accept(listensock,(struct sockaddr*)NULL,NULL);
send(connsock,buff,sizeof(buff),0);
recv(connsock,resbuff,1,0);
printf("%c\n",resbuff[0]);
/*下面就是执行控制命令*/
while(resbuff[0]!='q')
{
switch(resbuff[0])
{
case'a':
printf("start the control module.\n");
/*system 函数中的命令为模块编译命令，需链接到内核*/
system("gcc -O2 -g -Wall -DMODULE -D__KERNEL__
-I/usr/src/linux-2.4/include -c nrgcc.c");
system("gcc -O2 -g -Wall -DMODULE -D_KERNEL_ -I/usr/src/linux-2.4/include -c lgcc.c");
system("gcc -O2 -g -Wall -DMODULE -D__KERNEL__ -I/usr/src/linux-2.4/include -c nrgccf.c");
break;
…………/*其他控制命令省略*/
default:
printf("--------------------.\n");
break;
}
recv(connsock,resbuff,1,0);
}
…………
close(connsock);       /*通信完毕，关闭套接字*/
close(listensock);
}
```

由于控制端的网络管理系统是一个完整的可执行程序，系统的自动化要求比较高，为了方便使用，更是为了使程序有比较好的可移植性，把模块嵌入到程序中自动加载是关系到程序的可用性的一个重要部分。为此，在程序中调用 system()函数是一个很好的解决办法。

system()函数以系统命令作为其参数，并能将参数传递给系统 shell 进行解释、执行。例如

在上面的代码中，system()直接将 gcc 的编译命令(将在 17.3.4 小节介绍这个编译命令)作为其参数，这样在程序运行时，就能自动对 3 个控制模块进行编译了。

17.3 用 Netfilter 设计控制端功能模块

基于 Netfilter 的机制，便可以成功地在 Linux-2.4.20-8 内核下开发出一套简单的网络管理控制模块了。这些模块通过程序发出的控制指令进行动态的插入和卸载，分别实现了对 ICMP 网络数据，HTTP 站点，FTP 服务器站点的管理控制。

17.3.1 ICMP 管理控制模块

在 Linux-2.4 系统下可以使用 Netfilter 实现过滤数据报的功能模块，因此在控制管理 ICMP 数据报的模块中也同样可以使用 Netfilter。本小节向读者介绍 ICMP 管理控制模块的主要代码实现，关于详细的源代码，读者可参见光盘中的内容，本模块的代码位于光盘的/src/chapter_16/lgcc.c 文件。

首先在程序的开头，需要通知编译器把这个模块作为内核代码而不是普通的用户代码来编译，并确定模块代码的类型。源代码如下：

```
#ifndef __KERNEL__      /*按照内核模块编译*/
#define __KERNEL__
#endif
#ifndef MODULE          /*按照设备驱动程序模块编译*/
#define MODULE
#endif
```

ICMP 数据报处理函数 test_firewall()，函数原型定义如下：

```
static unsigned int test_firewall (unsigned int hooknum, struct sk_buff **skb,
const struct net_device *in ,const struct net_device *out,
int(*okfn)(struct sk_buff*));
```

在函数的参数列表中，hooknum 是内核指定的 5 个钩子点之一。第二个参数 skb 是一个指向指针(这个指针指向 sk_buff 类型的结构体)的指针，它是网络堆栈用来描述数据包的结构体，这个结构体定义在 linux/skbuff.h 中，由于这个结构体的定义很大，这里只着重于其中我们感兴趣的一些域。

sk_buff 结构体中最有用的域就是其中的 3 个联合了，这 3 个联合描述了传输层的头信息(例如 UDP、TCP、ICMP、SPX)、网络层的头信息(例如 IPv4、IPv6、IPX、RAW)和链路层的头信息(Ethernet 或者 RAW)。3 个联合相应的名字分别为 h、nh 和 mac。根据特定数据包使用的不同协议，这些联合包含了不同的结构体。

第三个参数 net_devices 结构体是 Linux 内核用来描述各种有效的网络接口的。第一个结构体 in 代表了数据包将要到达的接口，out 就代表了数据包将要离开的接口。最后一个参数是一个名为 okfn 的指向函数的指针，这个函数有一个 sk_buff 的结构体作为参数，返回一个整型值。

test_firewall()函数处理 ICMP 数据报的流程如图 17.9 所示。

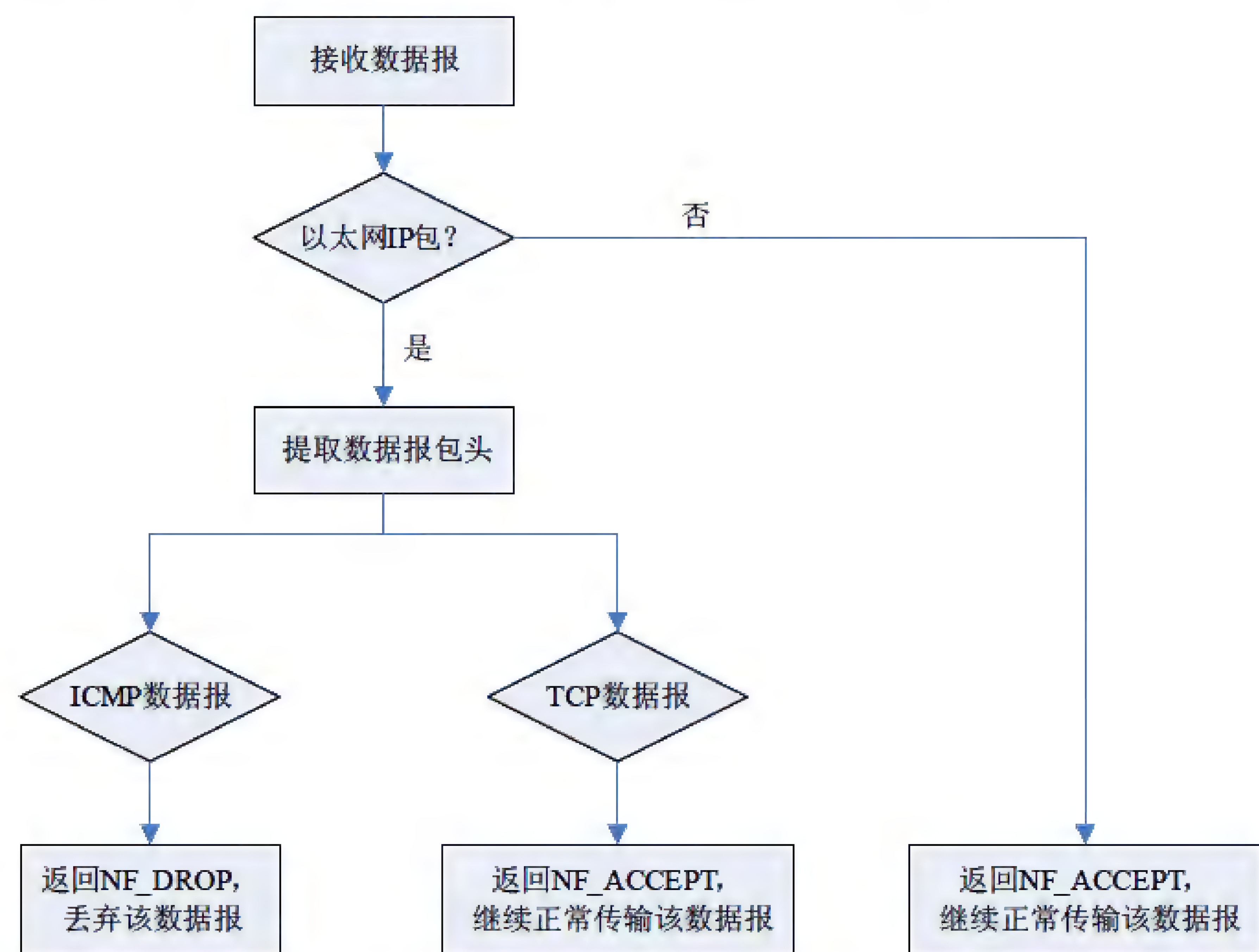


图 17.9　ICMP 数据报处理流程图

下面给出了 test_firewall()函数部分代码的实现及注释：

```
struct iphdr *iph;      /*定义一个指向 iphdr 结构体的指针 iph*/
…………
if((*skb)->protocol==htons(ETH_P_IP))
{
/*下面是取出数据报的 IP 头，检查是不是 ICMP 数据报类型*/
iph=(*skb)->nh.iph;
        if (iph->protocol==IPPROTO_ICMP)
{  /*如果是 ICMP 数据报，则返回 NF_DROP，让 Netfilter 丢弃该数据报*/
       printk("\nDROP a ICMP Packet");
return NF_DROP;
}
if(iph->protocol==IPPROTO_TCP)
{  /*如果是 TCP 数据报，则返回 NF_ACCEPT，让 Netfilter 继续传输该数据报*/
printk("\nPermit a valid access");
return NF_ACCEPT;
}
}
/*若是其他网络协议的数据报，则直接返回 NF_ACCEPT，让 Netfilter 继续传输该数据报*/
return NF_ACCEPT;
………
```

接下来，需要将我们设置处理 ICMP 数据报的信息初始化登记到防火墙用到的关键数据结构 nf_hook_ops 中，程序中定义为 struct nf_hook_ops ipl1 结构体，上面的处理函数 test_firewall()定义在钩子点 NF_IP_PRE_ROUTING 上，使用 IPv4 协议。这部分的代码实现及注释如下：

```
static struct nf_hook_ops ipl1=
{
{NULL,NULL},
test_firewall,         /*指明 HOOK 点的处理函数 test_firewall()*/
PF_INET,               /*IPv4 协议，此处也可以是 AF_INET(参考第 12 章)*/
NF_IP_PRE_ROUTING,     /*指在完整性校验之后，路由决策之前处理数据报*/
NF_IP_PRI_FILTER -1    /*运行的优先级*/
};
```

另外，还需要定义模块的初始化函数和模块卸载函数，程序中这两个函数分别定义为 init_module()和 cleanup_module()函数。

init_module()函数主要是调用 HOOK 的注册函数 nf_register_hook()，cleanup_module()主要是调用 HOOK 的注销函数 nf_unregister_hook()，调用它们时只需要将上面定义的防火墙关键数据结构 ipl1 作为唯一的参数传递过去。

而事实上，nf_register_hook()和 nf_unregister_hook()两个函数都是 Netfilter 框架中的函数原型，定义于内核的 net/core/netfilter.c 文件(分别位于 Line60 和 Line76)。有兴趣的读者可以查看它们的源代码。

17.3.2 FTP 管理控制模块

使用 Netfilter 同样可以通过对 TCP 21 和 20 端口的操作设计出控制和管理 FTP 服务器的模块。本小节向读者介绍 FTP 管理控制模块的主要代码实现，本模块的源代码位于光盘的/src/chapter_17/nrgccf.c 文件。

首先需要定义两个端口，即用于 FTP 服务的 21 和 20 端口，程序中的定义如下：

```
unsigned char *deny_port1="\x00\x15";
unsigned char *deny_port2="\x00\x14";
```

FTP 数据报处理函数为 lwfw_hookfn()，函数原型定义如下：

```
unsigned int lwfw_hookfn(unsigned int hooknum, struct sk_buff **skb,
                         const struct net_device *in, const struct net_device *out,
                         int (*okfn)(struct sk_buff *));
```

从中可以看到，lwfw_hookfn()函数的参数列表与 ICMP 数据报的处理函数是相同的，这里不再赘述。

因为在 TCP/IP 中，TCP 数据报实际上是通过 IP 数据报封装后传输的，所以我们先取出 IP 数据报的头部，再用一个指针指向 IP 数据报的开始部分，即 TCP 数据报。用 IP 数据报封装的 TCP 数据报结构如图 17.10 所示。而事实上，在本书第 12 章的图 12.11 中已经给出了 IP 数据报的报头格式，这里再次给出 IP 数据报的完整封装格式，读者可进行对比。

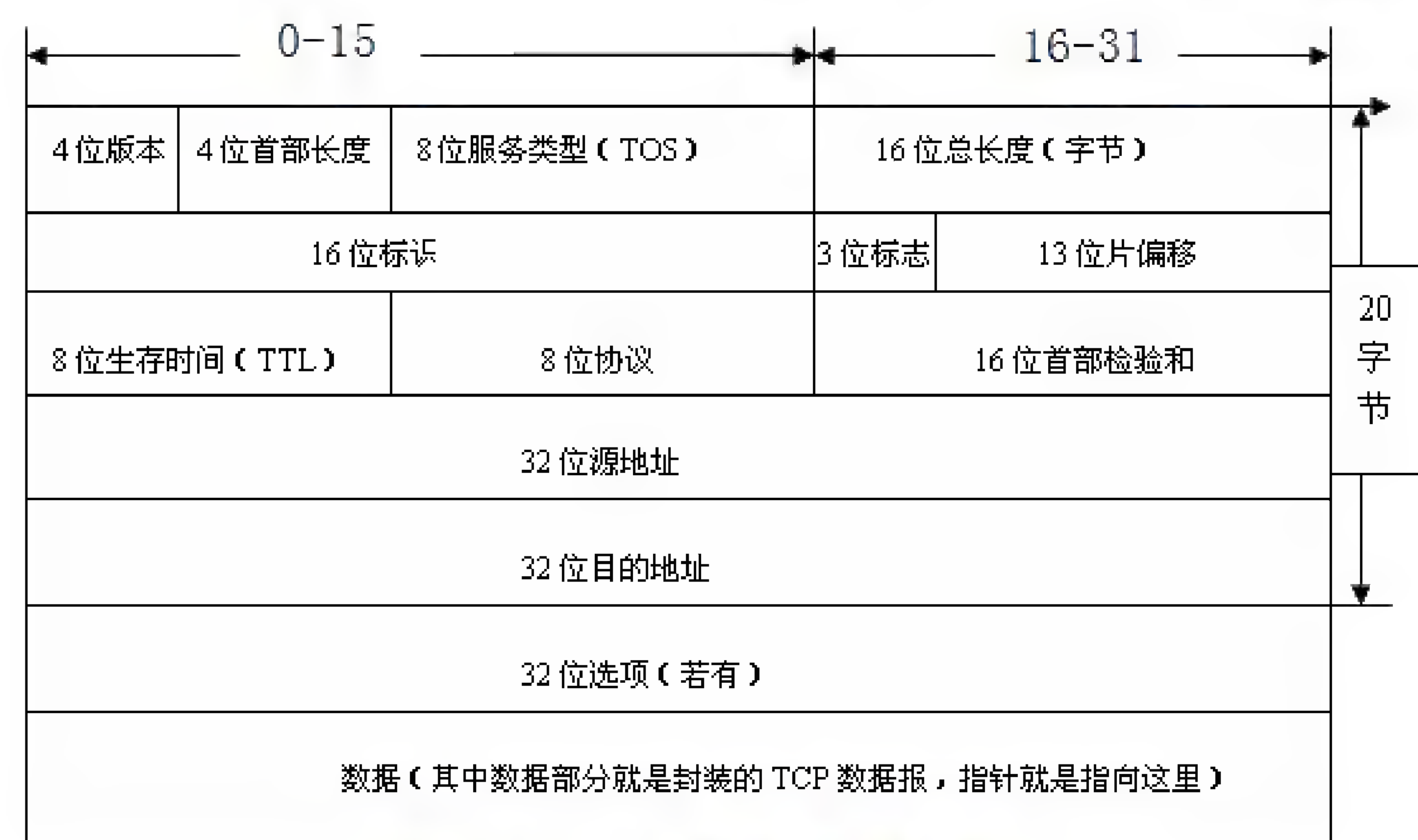


图 17.10　IP 数据报封装格式

将指针定位于 TCP 数据报的代码实现如下：

```
struct tcphdr *thead = (struct tcphdr *)(sk->data + (sk->nh.iph->ihl * 4));
```

lwfw_hookfn()函数处理 FTP 数据报的流程如图 17.11 所示。

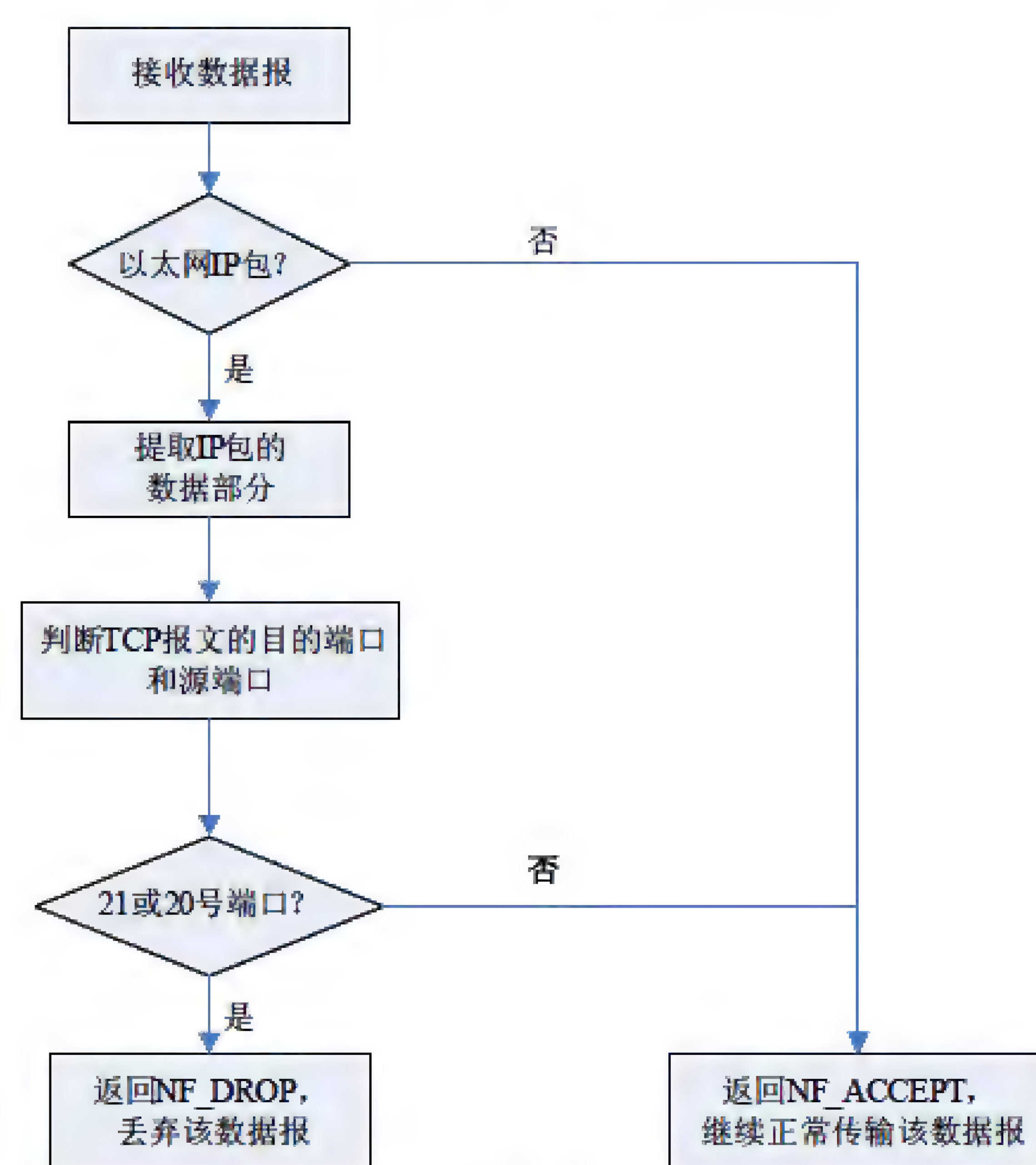


图 17.11　FTP 数据报处理流程图

此外，本部分的代码还包括模块的初始化函数 FTP_init()，模块卸载函数 FTP_exit()，以及登记、退出模块函数等。而模块的初始化和卸载函数依然是调用 Netfilter HOOK 的注册与注销函数，即 nf_register_hook()和 nf_unregister_hook()两个函数。

17.3.3　HTTP 管理控制模块

HTTP 的管理控制模块与 FTP 管理控制模块采用相同的设计思路，通过使用 Netfilter 控制 TCP 的 80 号端口，来管理 Web 网页(基于 HTTP 协议)的登录和访问数据报。本模块的源代码位于光盘中的/src/chapter_17/nrgcc.c 文件。

程序首先也是定义一个端口，即用于 HTTP 协议的 80 端口，程序中的定义如下：

```
unsigned char *deny_port = "\x00\x50";
```

此模块中，用于 HTTP 数据报的处理函数同样定义为 lwfw_hookfn()函数(注意这与 17.3.2 小节中的不是同一个函数，只是名字相同而已)，并且它的参数与前两个模块的处理函数也是相同的。

在此模块中，lwfw_hookfn()函数处理 HTTP 数据报的流程与 FTP 模块类似，只是将 Netfilter 的 HOOK 函数所检测的端口定义为 80 端口，如图 17.12 所示。

另外，关于模块处理函数的初始化、模块卸载函数，以及登记、退出模块函数等与 FTP 管理模块也是类同的，不再赘述。

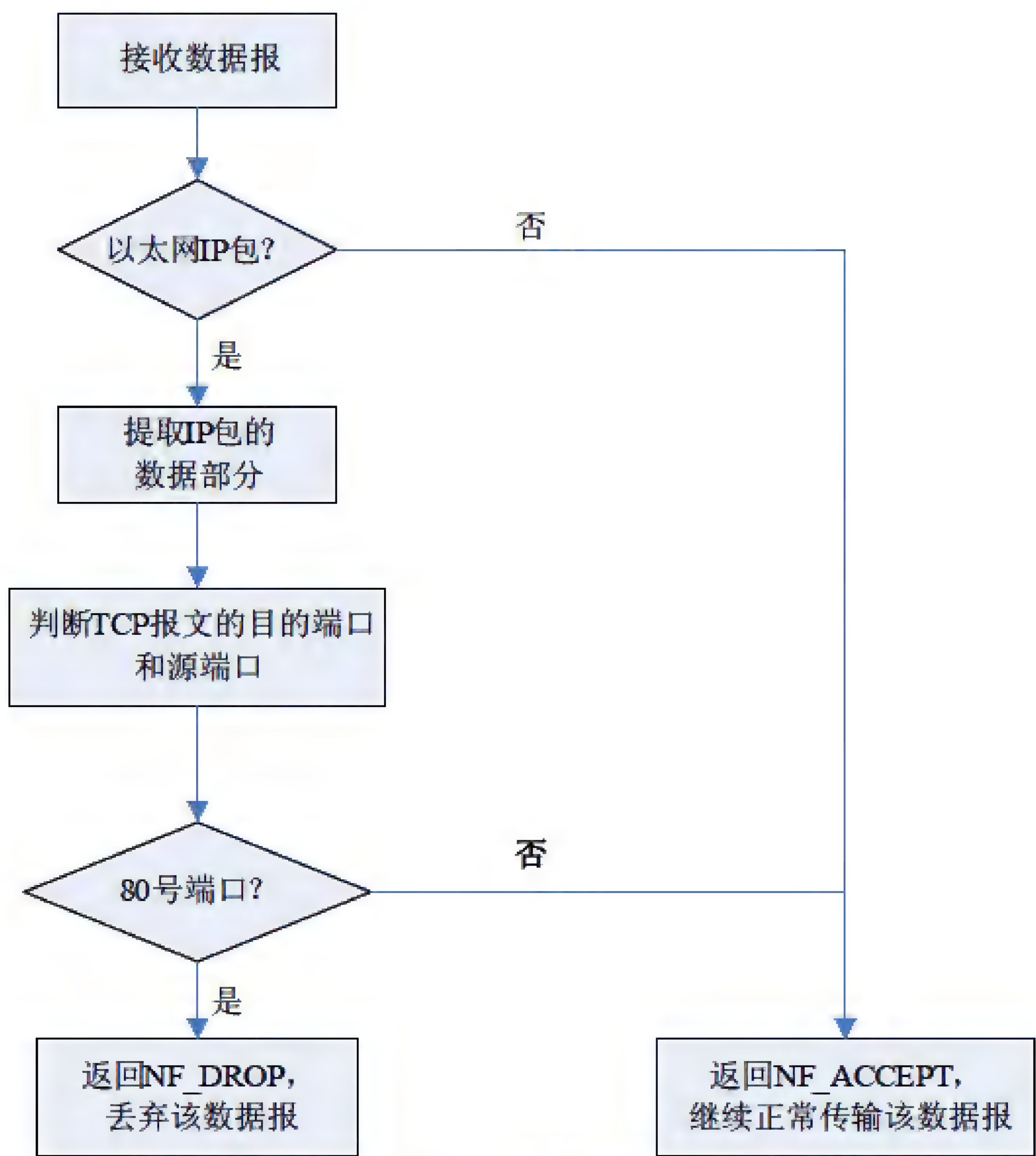


图 17.12　HTTP 数据报处理流程图

17.3.4　模块的编译、加载与卸载

在模块设计编写好后，必须将其编译成一个适合内核加载的对象文件。我们使用 gcc 编译器来编译程序，如果需要通知编译程序把这个模块作为内核代码而不是普通的用户代码来编译，则需要向 gcc 传递参数“-DMODULE”；若要对模块程序进行优化编译、链接，则就要用参数“-O2”；如果要对加载后的模块进行调试，那么就应该使用“-g”参数；同时需要使用“-Wall”参数将所有的警告信息显示出来，并使用“-c”开关通知编译程序在编译完这个模块文件后不调用链接程序。这样，便形成了编译模块文件的命令，格式如下：

```
gcc –O2 –g –Wall –DMODULE –D_KERNEL_ -I/usr/src/linux-2.4/include -c mymodules.c
```

命令行中 mymodules.c 为自己编写的模块程序源代码文件，/usr/src/linux-2.4/include 为计算机的内核版本。

接下来就是模块的加载，加载模块有两种方法：一种是通过 insmod 命令手工将模块载入

内核；另一种是根据需要载入模块(Demand Loaded Module)。当内核发现需要某个模块时，内核请求守护进程(kerneld)载入模块。守护进程是一个带有超级用户权限的普通用户进程。当该进程启动时，kerneld 开始执行，并为内核打开一个 IPC 通道。内核通过该通道发送消息，请求 kerneld 完成具体的任务。

需要注意的是，kerneld 的主要功能是加载和卸载内核模块，但 kerneld 自身并不执行这些任务，它只是通过某些程序(如 insmod、rmmod)来完成。kerneld 只是内核的代理，为内核进行调试。

另外，为了在程序中使各个模块能够自动加载至内核，以及从内核卸载，需要用到以下几个命令：

- lsmod：列举当前内核中已经加载的模块。
- insmod：把某个模块加载到内核中。
- rmmod：把当前某个未用的模块从内核中卸载。
- depmod：制造模块的信赖文件，以告诉将来的 insmod 在哪里找到模块来安装。

详细的情形，读者可参见程序的源代码。例如在程序的结束部分我们使用以下代码：

```
system("rmmod nrgcc ");
system("rmmod nrgccf ");
system("rmmod lgcc ");
```

自动卸载插入到内核中的 3 个模块，恢复操作系统的原来功能。

说明

解决模块编译的环境问题：

在开始程序设计的初期，只要是涉及内核模块的编译就会报出大量的错误，包括一些常用头文件的引导出错。原因是内核模块的编译需要在 Linux 操作系统中安装内核代码数据包 kernel-source-2.4.20-8.i386.rpm。下面简要向读者介绍笔者在设计过程中的解决方法。

Linux 内核代码数据包 kernel-source-2.4.20-8.i386.rpm 在 Red Hat Linux 9.0 的第二张安装光盘中。当然，也可以从因特网上(http://www.kernel.org)下载。

进入 kernel-source-2.4.20-8.i386.rpm 所在的目录：

```
#cd /mnt/cdrom/RedHat/RPMS/。
```

解压安装内核代码数据包：

```
#rpm -ivh /mnt/cdrom/RedHat/RPMS/kernel-source-2.4.20-8.i386.rpm。
```

查看 kernel-source 软件包的安装结果：

```
#rpm -q kernel-source
#cat /etc/grub.conf，看到/* “Hed Hat Linux(2.4.20-8)” */
#uname –r，/*查看新的内核的版本*/
```

17.4 软件功能测试

将光盘中的源代码复制至 Linux 主机下的某一个目录(一般是用户的主目录)，首先打开一个 shell 终端，进入软件源代码所在的目录，运行控制端程序，可以看到控制端的 TCP 端口处于监听的状态，命令如下(fubs 是编译生成的控制端的可执行文件)：

```
#./fubs
```

此时打开另一个 shell 终端，进入软件源代码所在目录，运行管理端程序：

```
#./keg localhost
```

keg 是编译生成的控制端的可执行文件，localhost 表示请求连接到本机的控制端。此时会看到在管理端的 shell 输出了一个“hello”字符串，表明管理端与控制端已成功建立 TCP 连接。

管理端打印输入命令提示信息，此时用户便可以输入字符进行相应的网络数据报控制操作了，管理端 shell 打印信息如下：

```
#./keg localhost
hello

please input the message:
please input the 'a' to add the module.
please input the 'c' close the WEB servers.
please input the 'o' open the WEB servers.
please input the 'f' filter the ICMP packet.
please input the 'i' accept the ICMP packet.
please input the 't' close the FTP servers.
please input the 'p' start the FTP servers.
please input the 'r' rmmod the all modules.
```

输入字符“a”，控制端便会自动调用 3 个 system()函数，动态编译生成 3 个管理控制模块。

管理控制模块生成后，便可以验证各个模块的功能了。当输入“f”时，控制端自动动态加载运行 ICMP 管理控制模块 lgcc.o，过滤 ICMP 数据报。此时可以打开第三个 shell 终端，运行命令(119.75.217.30 为百度网站的 IP 地址)：

```
ping 119.75.217.30
```

会看到系统一直得不到响应，这说明 ICMP 数据报发送超时。请读者参考第 12 章中最后的那个 ping 程序，可以知道 ICMP 数据报文是如何被发送和接收的。

接着，在第二个 shell(管理端 shell)输入“i”字符命令，控制端运行 lgcc.o 模块，接收 ICMP 数据报。此时在第三个 shell 中再次使用“ping 119.75.217.30”命令，可以看到，ICMP 数据报接收成功。

接下来验证 FTP 管理控制模块功能，当在管理端 shell 输入字符命令“t”时，控制端自动动态加载运行 FTP 管理控制模块 nrgccf.o，过滤 FTP 数据报，也即关闭 FTP 服务。仍然是在第

三个 shell 来验证，输入如下命令：

```
# ftp 140.186.70.20
```

此时可以看到 FTP 连接请求一直得不到响应。管理端 shell 输入“p”，打开 FTP 服务，并再次运行第三个 shell 下的命令，得到输出：

```
# ftp 140.186.70.20
Connected to 140.186.70.20 (140.186.70.20).
220 GNU FTP server ready.
Name (140.186.70.20:root):
```

表明 FTP 连接请求已经建立成功，输入用户名和密码，便可以向 FTP 服务器进行资源的下载和上传了。(参考第 12 章中的程序 12.4，140.186.70.20 是 GNU 的 FTP 服务器地址。)

HTTP 管理控制模块是实现对访问 Web 网页数据报的控制。在第二个 shell(管理端 shell)输入字符命令“c”，控制端自动动态加载运行 HTTP 管理控制模块 nrgcc.o，过滤 HTTP 数据报，也即关闭 Web 服务器。

此时在浏览器的地址栏输入以下地址：http://www.baidu.com(事实上，可以输入任何网站的网址)，会看到系统弹出失败信息对话框：“当试图与 www.baidu.com 联系时，操作超时”。这说明系统此时无法接收 HTTP 数据报，也就无法打开任何的 Web 网页。

再次输入字符命令“o”，控制端再次调用 nrgcc.o 模块，接收 HTTP 数据报，也即开启 Web 服务器。此时 Linux 的浏览器便可以访问任何的合法网址了。

另外，还可以使用“r”字符命令卸载所有的模块，使 Linux 内核恢复原来的工作机制。

最后，给出测试过程中两个 shell 终端的输出信息，帮助读者理解源代码和模块的工作机制。

管理端的全部打印信息如下：

```
#./keg localhost
hello

please input the message:
please input the 'a' to add the module.
please input the 'c' close the WEB servers.
please input the 'o' open the WEB servers.
please input the 'f' filter the ICMP packet.
please input the 'i' accept the ICMP packet.
please input the 't' close the FTP servers.
please input the 'p' start the FTP servers.
please input the 'r' rmmod the all modules.
a
----------------------.
f
start filter the ICMP packet.
----------------------.
i
start receive the ICMP packet.
----------------------.
```

```
t
start close the FTP servers.
------------------------.
p
start open the FTP servers.
------------------------.
c
start close the WEB server.
------------------------.
o
start open the WEB server.
------------------------.
r
rmmod the control modules.
------------------------.
```

控制端的全部打印信息如下：

```
#./fubs
a
start the control module.
---------------------.
filter the ICMP packet.
---------------------.
accept the ICMP packet.
---------------------.
close the FTP servers.
---------------------.
start the FTP servers.
---------------------.
close the 80 source port.
---------------------.
open the 80 source port.
---------------------.
all the control module has rmmod.
rmmod: module nrgcc is not loaded
rmmod: module nrgccf is not loaded
rmmod: module lgcc is not loaded
---------------------.
```

17.5 本章小结

本章首先介绍了 Linux 下新一代防火墙构建平台 Netfilter 的工作原理，在此基础上讲解了本章软件的设计流程，以及一些关键的实现细节。软件实现了对固定端口，网页访问，以及不同协议类型的数据报文的管理和控制，基本实现了防火墙的简单功能。通过本章的学习，读者应该深入理解了如何使用 Netfilter 这个网络数据包过滤机制编写网络数据的控制程序。

第18章

基于Linux的嵌入式家庭网关远程交互操作平台的设计

计算机技术的快速发展和网络技术的日新月异，给嵌入式计算机系统带来了巨大的发展机会。目前，嵌入式系统已经渗透到各个领域，得到了广泛的应用。32 位嵌入式 RISC 处理器 ARM 的应用已扩展到世界范围，占据了低功耗、低成本和高性能的嵌入式系统应用领域。同时，嵌入式 Web 技术将是组建基于 Internet 的远程分布式测控系统的关键技术之一，它为 Web 技术渗透到测控领域及通讯仪器领域发挥了重大作用，将是信息领域的一次重大的革新。

本章课题选用三星公司的基于 ARM920T 核的 S3C2410X 芯片，在 Linux 操作系统和 ARM9 的软硬件平台上利用嵌入式 Web 服务器 Boa，结合 CGI 技术，讲述了嵌入式家庭网关远程交互操作平台的设计与实现。课题中嵌入式网关和智能家电设备通过 RS485 总线标准互连，使得用户能够通过浏览器远程访问和控制家庭智能设备。限于试验条件，课题选用了两片 MSC-51 单片机来模拟家庭智能设备。该嵌入式网关实现了远程分布式测控和通讯，性价比高，具有很大的应用价值，可以应用于工业控制、交通控制等领域；同样对于家庭范围内也是适用的，此时嵌入式网关可抽象称为嵌入式家庭网关。



本章内容：

◎ 嵌入式技术简介。
◎ 家庭网关的概念及其网络体系结构。
◎ 嵌入式家庭网关的开发平台。
◎ 远程交互平台的设计。
◎ Linux 下软件模块的具体实现。

18.1 嵌入式技术简介

嵌入式系统无疑是当今最热门的概念之一，但究竟什么是嵌入式系统？嵌入式系统有哪些重要组成部分？什么是嵌入式操作系统？嵌入式处理器又有哪些种类？本节将向读者介绍嵌入式技术中的相关基本概念。

18.1.1 嵌入式系统的概念

嵌入式系统(Embedded System)通常是面向特定应用的。目前嵌入式计算机系统比较正式的定义为：以应用为中心，以计算机技术为基础，软件硬件可裁减，符合应用系统对功能、可靠性、成本、体积、功耗严格要求的专用计算机系统。嵌入式系统一般由嵌入式微处理器、外围硬件设备、嵌入式操作系统及用户的应用程序 4 个部分组成，用于对其他设备的控制、监视或管理等功能。

嵌入式系统是将先进的计算机系统、半导体技术和电子技术与各个行业的具体应用相结合的产物。这决定了它必然是一个技术密集、资金密集、高度分散、不断创新的知识集成系统，与通用 PC 机比较，嵌入式系统的突出特点是专用性、成本敏感性及较高的可靠性。嵌入式系统对系统软件和应用软件的要求也与通用计算机有所不同，一般要求如下：

- 软件要求固化存储。
- 许多应用要求系统软件具有实时处理能力。
- 多任务操作系统是知识集成的平台，也是走向工业标准化道路的基础。

通用计算机系统具有完善的操作系统和应用程序接口(Application Programming Interface，API)，应用软件直接在操作系统平台上运行。嵌入式系统则不同，应用程序可以没有操作系统直接在芯片上运行，但为了合理地调度多任务，利用系统资源、系统函数及专用接口函数，用户必须自行选配嵌入式操作系统(Embedded Operating System，EOS)开发平台，这样才能保证程序执行的实时性和可靠性，并减少开发时间，提高软件质量。一个优秀的 EOS 是嵌入式系统成功的关键。

近年来，随着计算机技术、通信技术的飞速发展，嵌入式系统已经广泛渗透到人们的工作、生活中，如家用电器、手持通信设备、信息终端、仪器仪表、汽车、航天航空、军事装备、制造工业、过程控制等。今天，嵌入式系统带来的工业年产值已超过 1 万亿美元。据统计，嵌入式处理器的数量占分散处理器的 94%，而 PC 机用的处理器只占 6%。根据美国嵌入式系统专业杂志 RTC 报道，21 世纪初的 10 年中，全球嵌入式系统市场需求量具有比 PC 市场大 10～100 倍的商机。

另一方面，Internet 现已成为世界上最重要的基础信息设施之一，Internet 使全球化的信息化交流变得非常容易。如果嵌入式系统能够连接到 Internet 上面，则可以方便、低廉地将信息传送到几乎世界上的任何一个地方。

可以预言，嵌入式设备与 Internet 的结合代表着嵌入式系统和网络技术的真正未来，它具有巨大的市场潜力，目前，包括 Siemens、Philips 和 Motorola 在内的数十家公司联合成立了“嵌入式 Internet 联盟(ETI)”，共同推动这一技术的发展。

18.1.2 嵌入式操作系统

从 20 世纪 80 年代开始，市场上出现了各种各样的商用嵌入式操作系统，这些操作系统大部分都是为专有系统开发的，从而逐步演化成了现在多种形式的商用嵌入式操作系统百家争鸣的局面。这些操作系统有嵌入式 Linux、Windows CE、VxWorks 和 pSOS 等。下面简单介绍嵌入式操作系统的概念和一些典型的嵌入式操作系统。

1. 定义及特点

嵌入式操作系统(EOS)是一种实时的、支持嵌入式系统应用的操作系统软件，它是嵌入式系统(包括硬、软件系统)的核心，通常包括与硬件相关的底层驱动软件、系统内核、设备驱动接口、通信协议、图形界面、标准化浏览器 Browser 等。

与通用操作系统比较，嵌入式操作系统具有如下特征：

(1) 小巧。嵌入式系统所能够提供的资源有限，所以嵌入式操作系统必须做到小巧，以满足嵌入式系统硬件的限制。

(2) 实时性。目前，大多数 EOS 都具有 RTOS 内核，Linux、Windows CE 的实时性较弱，但改进后的 Linux 系统(如 RT- Linux)实时性也很强。

(3) 强稳定性与高可靠性。任务管理与调度策略能保证操作系统上的应用程序可靠运行。

(4) 可移植性。大部分嵌入式操作系统(EOS)可以应用于多种嵌入式处理器，如 MPU、MCU、DSP、ARM、PPC 等平台上。

(5) 可裁减。嵌入式操作系统可以根据应用需要进行裁减，去掉多余的部分，或者简化相应的模块。

(6) 可固化代码。嵌入式系统中的存储空间有限，操作系统代码与应用软件代码通常需要被固化在系统的 ROM 中。

2. 典型的嵌入式操作系统

典型的嵌入式操作系统包括嵌入式 Linux、Windows CE、VxWorks 和 pSOS 等。

(1) 嵌入式 Linux

在所有的操作系统中，嵌入式 Linux 的发展最快，应用也是最广泛的。嵌入式 Linux 本身的种种特性使其成为嵌入式开发的首选。在进入市场的前两年中，嵌入式 Linux 的设计通过广泛应用而获得巨大成功。随着嵌入式 Linux 技术的成熟，定制需要的尺寸尤为方便，同时支持更多平台，并从早期的试用阶段迈进到成为嵌入式市场的主流。

嵌入式 Linux 的最大特点是代码的开放性。代码的开放性是与后 PC 时代的智能设备的多样性相适应的。代码的开放性主要体现在源代码可获得上。Linux 代码开发就像是“集市式”开发，任意选择并按自己的意愿整合出新的产品。嵌入式 Linux 技术的普及发展，为国内单片机工程师在软件功能方面提供了极大的支持，为软件引入了 TCP/IP 网络特性，引入了软件操作系统的健壮性，这都极大地增加了系统的功能和提高了系统的性能。

(2) Windows CE

Windows CE 是微软公司的产品，它采用模块化设计，并允许对于从掌上电脑到专用的工控电子设备进行定制。操作系统的基本内核需要至少 200KB 的 ROM。从 SEGA 的 DreamCast 游戏机到现在大部分的高价掌上电脑都采用了 Windows CE 作为操作系统，其缺点是价格过高，使得整个产品的成本急剧上升。

(3) VxWorks

VxWorks 操作系统是美国 WindRiver 公司于 1983 年设计开发的一种实时操作系统。它是专门为实时嵌入式系统设计开发的操作系统软件，为程序员提供了高效的实时任务调度、中断管理、实时的系统资源及实时的任务间通信。应用程序员可以将尽可能多的精力放在应用程序本身，而不必再去关心系统资源的管理。VxWorks 拥有良好的持续发展能力，高性能的内核及友好的用户开发环境。它以良好的可靠性和卓越的实时性被广泛地应用于通信、军事、航空航天等高精尖技术，以及实时性要求较高的领域中。它是目前嵌入式系统领域中使用最广泛、市场占有率最高的系统。谁都不能否认 VxWorks 是一个非常优秀的实时系统，但其昂贵的价格使不少用户望而却步。

(4) pSOS

pSOS 是 ISI(Integrated Systems，Inc.)公司研发的产品。该公司成立于 1980 年。pSOS 是一个模块化、高性能、完全可扩展的实时操作系统，它提供了一个完全多任务环境，在定制的或是商业化的硬件上提供高性能和高可靠性。它包括单处理器支持模块(pSOS+)、多处理器模块(pSOS+m)、文件管理器模块(pHILE)、TCP/IP 通信包(pNA)、流式通信模块(OpEN)、图形界面、Java、HTTP 等。

18.1.3 嵌入式处理器

从硬件方面来讲，各式各样的嵌入式处理器是嵌入式系统硬件中的最核心的部分，而目前世界上具有嵌入式功能特点的处理器已经超过 1000 种，流行体系结构包括 MCU、MPU 等 30 多个系列。鉴于嵌入式系统广阔的发展前景，很多半导体制造商都大规模地生产嵌入式处理器，并且公司自主设计处理器也已经成为未来嵌入式领域的一大趋势，其中从单片机、DSP 到 FPGA 有着各式各样的品种，速度越来越快，性能越来越强，价格也越来越低。目前嵌入式处理器的寻址空间可以从 64KB 到 16MB，处理速度最快可以达到 2000MIPS，封装从 8 个引脚到 144 个引脚不等。

根据其现状，嵌入式处理器可以分成下面几类。

(1) 嵌入式微处理器(Micro Processor Unit，MPU)

嵌入式微处理器是由通用计算机中的 CPU 演变而来的。它的特征是具有 32 位以上的处理器，具有较高的性能，当然其价格也相应较高。但与计算机处理器不同的是，在实际嵌入式应用中，只保留和嵌入式应用紧密相关的功能硬件，去除其他的冗余功能部分，这样就以最低的功耗和资源实现嵌入式应用的特殊要求。和工业控制计算机相比，嵌入式微处理器具有体积小、重量轻、成本低、可靠性高的优点。目前主要的嵌入式处理器类型有 Am186/88、386EX、SC-400、Power PC、68000、MIPS、ARM/StrongARM 系列等。其中 ARM/StrongARM 是专为手持设备开发的嵌入式微处理器，属于中档的价位。

(2) 嵌入式微控制器(Micro Controller Unit，MCU)

嵌入式微控制器的典型代表是单片机，从 70 年代末单片机出现到今天，虽然已经经过了 20 多年的历史，但这种 8 位的电子器件目前在嵌入式设备中仍然有着极其广泛的应用。单片机芯片内部集成了 ROM/EPROM、RAM、总线、总线逻辑、定时/计数器、看门狗、I/O、串行口、脉宽调制输出、A/D、D/A、Flash RAM、EEPROM 等各种必要功能和外设。和嵌入式微处理器相比，微控制器的最大特点是单片化，体积大大减小，从而使功耗和成本下降、可靠性提高。

微控制器是目前嵌入式系统工业的主流。微控制器的片上外设资源一般比较丰富，适合于控制，因此称为微控制器。

由于 MCU 低廉的价格，优良的功能，所以拥有的品种和数量最多，比较有代表性的包括 8051、MCS-251、MCS-96/196/296、P51XA、C166/167、68K 系列及 MCU 8XC930/931、C540、C541，并且有支持 I2C、CAN-Bus、LCD 及众多专用 MCU 和兼容系列。目前 MCU 占嵌入式系统约 70%的市场份额。近来 Atmel 出产的 Avr 单片机由于其集成了 FPGA 等器件，所以具有很高的性价比，势必将推动单片机获得更高的发展。

(3) 嵌入式 DSP 处理器(Embedded Digital Signal Processor，EDSP)

DSP 处理器是专门用于信号处理方面的处理器，其在系统结构和指令算法方面进行了特殊设计，具有很高的编译效率和指令的执行速度。在数字滤波、FFT、谱分析等各种仪器上 DSP 获得了大规模的应用。

DSP 的理论算法在 70 年代就已经出现，但是由于专门的 DSP 处理器还未出现，所以这种理论算法只能通过 MPU 等由分立元件实现。MPU 较低的处理速度无法满足 DSP 的算法要求，其应用领域仅仅局限于一些尖端的高科技领域。随着大规模集成电路技术发展，1982 年世界上诞生了首枚 DSP 芯片。其运算速度比 MPU 快了几十倍，在语音合成和编码解码器中得到了广泛应用。至 80 年代中期，随着 CMOS 技术的进步与发展，第二代基于 CMOS 工艺的 DSP 芯片应运而生，其存储容量和运算速度都得到了成倍提高，成为语音处理、图像硬件处理技术的基础。到 80 年代后期，DSP 的运算速度进一步提高，应用领域也从上述范围扩大到了通信和计算机方面。90 年代后，DSP 发展到了第五代产品，集成度更高，使用范围也更加广阔。

目前最为广泛应用的是 TI 的 TMS320C2000/C5000 系列，另外如 Intel 的 MCS-296 和 Siemens 的 TriCore 也有各自的应用范围。

(4) 嵌入式片上系统(System on Chip，SoC)

SoC 追求产品系统最大包容的集成器件，是目前嵌入式应用领域的热门话题之一。SoC 最大的特点是成功实现了软硬件无缝结合，直接在处理器片内嵌入操作系统的代码模块。而且 SoC 具有极高的综合性，在一个硅片内部运用 VHDL 等硬件描述语言，实现一个复杂的系统。用户不需要再像传统的系统设计一样，绘制庞大复杂的电路板，一点一点地连接焊制，只需要使用精确的语言，综合时序设计直接在器件库中调用各种通用处理器的标准，然后通过仿真之后就可以直接交付芯片厂商进行生产。由于绝大部分系统构件都是在系统内部，整个系统就特别简洁，不仅减小了系统的体积和功耗，而且提高了系统的可靠性，提高了设计生产效率。

由于 SoC 往往是专用的，所以大部分都不为用户所知，比较典型的 SoC 产品是 Philips 的 Smart XA。少数通用系列如 Siemens 的 TriCore，Motorola 的 M-Core，某些 ARM 系列器件，Echelon 和 Motorola 联合研制的 Neuron 芯片等。

18.2 家庭网关的概念及其网络体系结构

家庭网关(Home Gateway)是实现智能家庭网络的关键与核心。本章所要讲述的设计是要在嵌入式家庭网关研究的相关技术背景下，以开发经济实用的嵌入式家庭网关为研究目标，设计

基于 ARM&Linux 的嵌入式家庭网关系统。本节首先向读者简要介绍智能家庭网络的概念，以及家庭网关的网络体系结构。

18.2.1 智能家庭网络的概念

智能家庭网络(Intelligent Home Network)是集计算机、通信和消费技术于一体的 3C 系统，是后 PC 时代 IT 业的又一大热点，它是指在家庭内部通过一定的传输介质将各种电子设备、电气设备和电子系统连接起来，采用统一的通信协议，对内管理家庭内部网中智能家电的运作、协调；对外实现家庭内部网和 Internet、公用电话网或 GSM/GPRS 移动网络等公众通信平台的连接，支持远端对家庭内部设备的控制和监测。

一个完善的家庭网络应该包括高速数据通信、高速的音频/视频(A/V)信号传输和低速控制三部分。各部分通过一个类似网关的平台，即家庭网关，对外与互联网、有线/无线通信线路相连，对内将各部分的设备连接起来，实现各设备之间相互通信、数据交换、存储和控制。家庭网关是智能家庭网络物理上与逻辑上的核心。

家庭网关的实现目前主要有 PC 机与嵌入式系统两种方式。与 PC 机比较，嵌入式系统具有体积小、成本低、可靠性高、稳定性好及功耗低等优点，更符合家庭网关的性能要求。本章所要向读者介绍的设计目标即基于 ARM&Linux 的嵌入式家庭网关的实现。

家庭网关可以将智能家电连接到互联网，成为家庭内部网的出口。通过它可以对家庭内部网络中的家电、门窗等进行中央监控和远程监控，可以随时随地监视家中的安全情况，可以自动报警等。人们可以通过使用远程监控软件，在离家很远的地方通过 Internet 远程地操作和控制家用电器。通过家庭网关，水、电、煤气表可以进行自动抄表和自动结算，而省去繁琐的人工抄表。另外，家庭网关还具有上网功能，可以用于上网浏览、收发电子邮件、发布个人主页、参加网上论坛等。

家庭的数字化、智能化、网络化和信息化一方面为社会信息化解决了最基本的单元问题，另一方面也为信息技术的发展提供了一个新型的方法和领域，同时也促进了家电设备数字化、传感器多样化及网络互联技术的发展。同时作为社会的基本组成单元，家庭的信息化无疑是整个社会信息化的最重要标志，对社会和谐发展、科技进步和经济繁荣都有着极为重要的意义。

18.2.2 家庭网关的远程交互操作技术简介

一个智能家庭网关可以看成是一个信息处理系统，组成系统的各单元就是连接在网络各节点的设备。家庭网关一方面辅助不具备信息化条件的设备实现信息化，即提供信息处理的能力；另一方面又提供统一的信息交换接口及控制规则，从而实现从内部家庭网络到 Internet 的信息互通。目前实现信息管理和协议转换的方式主要有两种：一种是 Browser+Web Server+CGI 技术(B/S 架构)；另一种是监控软件+应用服务器+现场总线驱动技术(C/S 架构)。

本章的设计基于 B/S 远程交互操作技术模式，客户端只需要单一的浏览器软件，其他大量工作都由 Web Server (也就是家庭网关)完成。这种模式使用简单、易于维护、扩展性好、软件升级只要在服务器端进行即可。但是这种方案也有其局限性，那就是在这种方式下通常都只是通过浏览器对家庭网络中的单个智能节点进行一对一的监控，效率不够高。另外，B/S 结构本质上也是一种 C/S 结构，它是三层 C/S 结构在 Web 上应用的特例。在 B/S 系统中，客户机的工作得以极大地减轻，但服务器将负担更多的工作，对数据库的访问和应用程序的执行将在服

务器上完成。

18.2.3 嵌入式家庭网关的网络体系结构

家庭网关是实现家庭内部网络设备与 Internet 网络互联的通信平台，嵌入式家庭网关的研究中涉及许多关键技术，如网络体系结构的研究、网关通信协议、嵌入式操作系统(EOS)和嵌入式系统的实现等。要实现嵌入式家庭网关首先要确立其网络体系结构。

家庭内部网络是将家庭中所有电子设备连接成一个由智能软件管理的网络。通过对家庭生活中常用电器具有的信息特征的分析得出，家庭网络的信息组成可以分成 3 类，其中包括控制信息、A/V 音频视频信号、数字数据。冰箱、空调、洗衣机、微波炉等电器处理的都是控制信息，信息流量小，可以构成家电控制网络；电视、VCD/DVD 等电器之间使用 A/V 端子和 A/V 端子线相互连接，互相传输 A/V 信息，信号数据量大，构成家庭 A/V 网络；计算机、IP 电话、数字电视等设备处理的都是数字化数据，都支持 TCP/IP 协议，数据量大，可以构成家庭数据通信网络。3 个家庭内部子网通过家庭网关对外与 Internet 等公众网连接，对内实现各子网设备的互联与互访。

通过家庭网络与智能家电设备之间的相互关系，设计带有嵌入式家庭网关的家庭网络模型如图 18.1 所示。该模型分为 3 个部分：前端数据采集部分、家庭网络核心——嵌入式家庭网关系统、远程交互操作平台。其中前端数据采集部分的主要功能是对各种家庭电器进行数据采集，并将采集来的数据发送给核心模块——家庭网关，从而进行处理与保存。而远程交互操作平台的主要功能是利用客户端的普通 PC 机浏览器来获取家庭网关中的各种受控设备的信息。对于用户来说远程访问部分的内部实现方法具有透明性，其数据处理与维护工作都由家庭网关来完成。

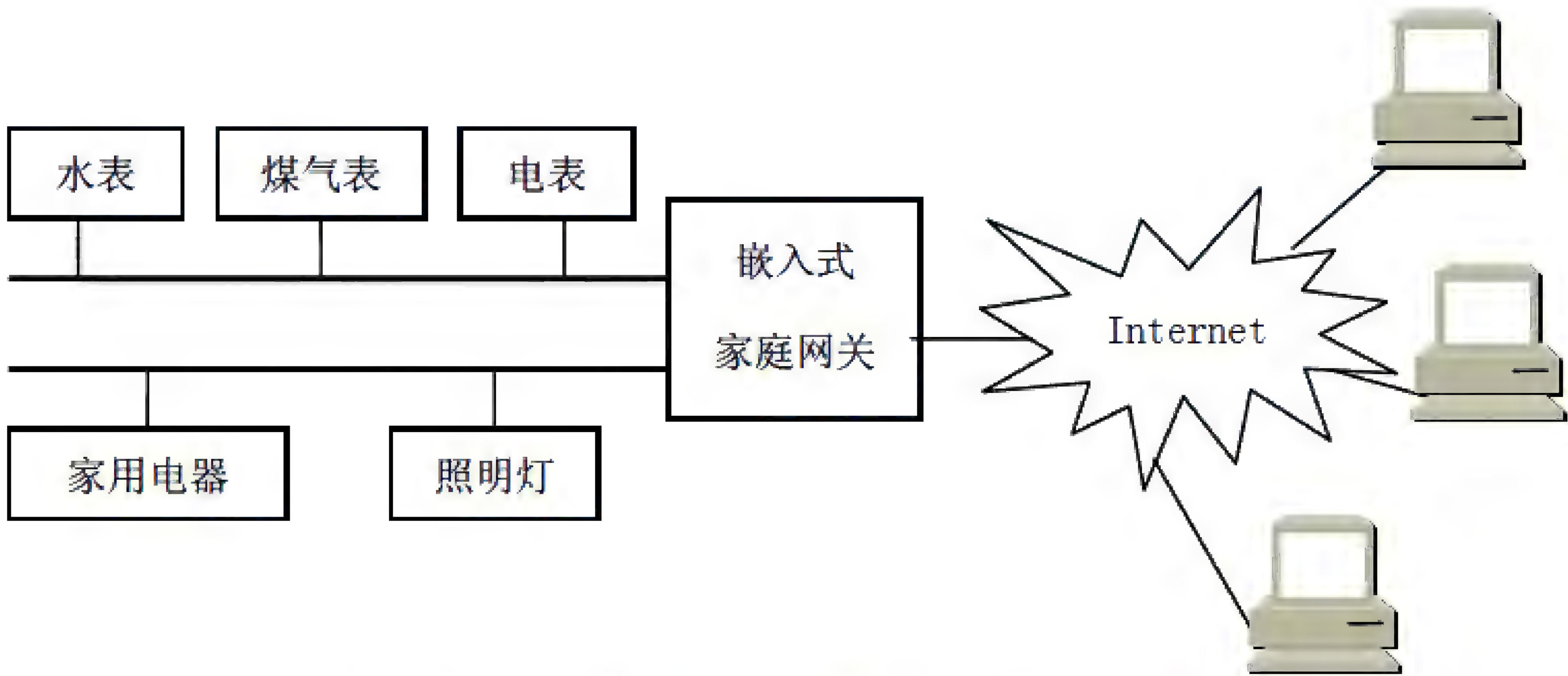


图 18.1 嵌入式家庭网关的 Internet 网络体系结构模型

嵌入式家庭网关作为家庭网络的核心和家庭内部设备与远程用户之间的桥梁，是整个家庭网络中的重点，也是本章的设计中研究的主要内容。

在通信协议上，家庭网关为了支持互联网功能，要运行 TCP/IP 协议，同时，还要运行控制子网通信协议支持家电控制网。控制子网通信协议是一种面向局域网的简单通信协议，结构

比 TCP/IP 协议要简单得多，容易实现，而且占用的内存空间小，可以简化硬件成本，比 TCP/IP 协议更适合家庭使用。根据市场反馈的信息，目前家庭网络系统在家庭安防控制、家庭对讲、灯光控制系统及家电设备的远程控制方面更具有现实的市场需求。因而，为了使家庭网关的设计更具备经济性及实用性，网关设计的性能指标将主要针对小信息流量的低速控制子网。

18.3 嵌入式家庭网关的开发平台

为缩短开发周期，降低开发的难度，增强系统的稳定性和功能的可扩充性，嵌入式软件的设计将建立在嵌入式操作系统的基础上，充分利用操作系统所提供的进程管理、文件管理、内存管理及网络功能，大大提高开发的效率，方便软件的维护和升级。本节向读者介绍家庭网关的开发平台的架设，包括硬件平台 ARM9 处理器 S3C2410X 的简单介绍，和适用于 S3C2410X 的交叉编译器 arm_linux_gcc 的构建。

18.3.1 S3C2410 微处理器简介

硬件平台是系统设计的基础，合理的硬件选择可以简化系统的设计，提高系统的可靠性、降低设计复杂度。本系统中的嵌入式家庭网关硬件平台选用的是由广州友善之臂科技公司生产的 S3C-2410X 开发板。

S3C-2410X 是一款基于 ARM9 的嵌入式计算机平台，它基于三星公司的 ARM 处理器 S3C2410X，采用 6 层板设计。S3C2410X 处理器使用 ARM920T 核，内部带有全性能的 MMU(内存处理单元)，它适用于设计移动手持设备类产品，具有高性能、低功耗、接口丰富和体积小等优良特性。S3C-2410X 正是基于此芯片本身的各种特点而设计的。

三星公司推出的 16/32 位 RISC(Reduced Instruction Set Computer，精简指令集计算机)处理器 S3C2410X，为手持设备和一般类型应用提供了低价格、低功耗、高性能小型控制器的解决方案。为降低整个系统的成本，S3C2410X 提供了以下丰富的内部设备：独立的 16KB 的指令 cache 和 16KB 数据 cache、MMU 虚拟存储器管理、LCD 控制器(支持 STN&TFT)、支持 NAND Flash 系统引导、系统管理器(片选逻辑和 SDRAM 控制器)、3 通道 UART、4 通道 DMA、4 通道 PWM 定时器、I/O 端口，RTC、8 通道 10 位 ADC 和触摸屏接口、IIC-BUS 接口、USB 主机、USB 设备、SD 主卡和 MMC 卡接口，2 通道的 SPI 及内部 PLL 时钟倍频器。现在，基于 ARM920T 核的微处理器已广泛应用于 PDA、移动通信、路由器、工业控制等领域，其内部框架图如图 18.2 所示。

S3C2410X 采用了 ARM920T 内核和 0.18um 工艺的 CMOS 标准宏单元和存储器单元。ARM920T 核有两种工作模式：32 位指令集 ARM 模式和 ARM 模式的重编码子集(16 位指令集)Thumb 模式。使用 Thumb 指令集取代 ARM 指令集可以得到更高的代码密度。

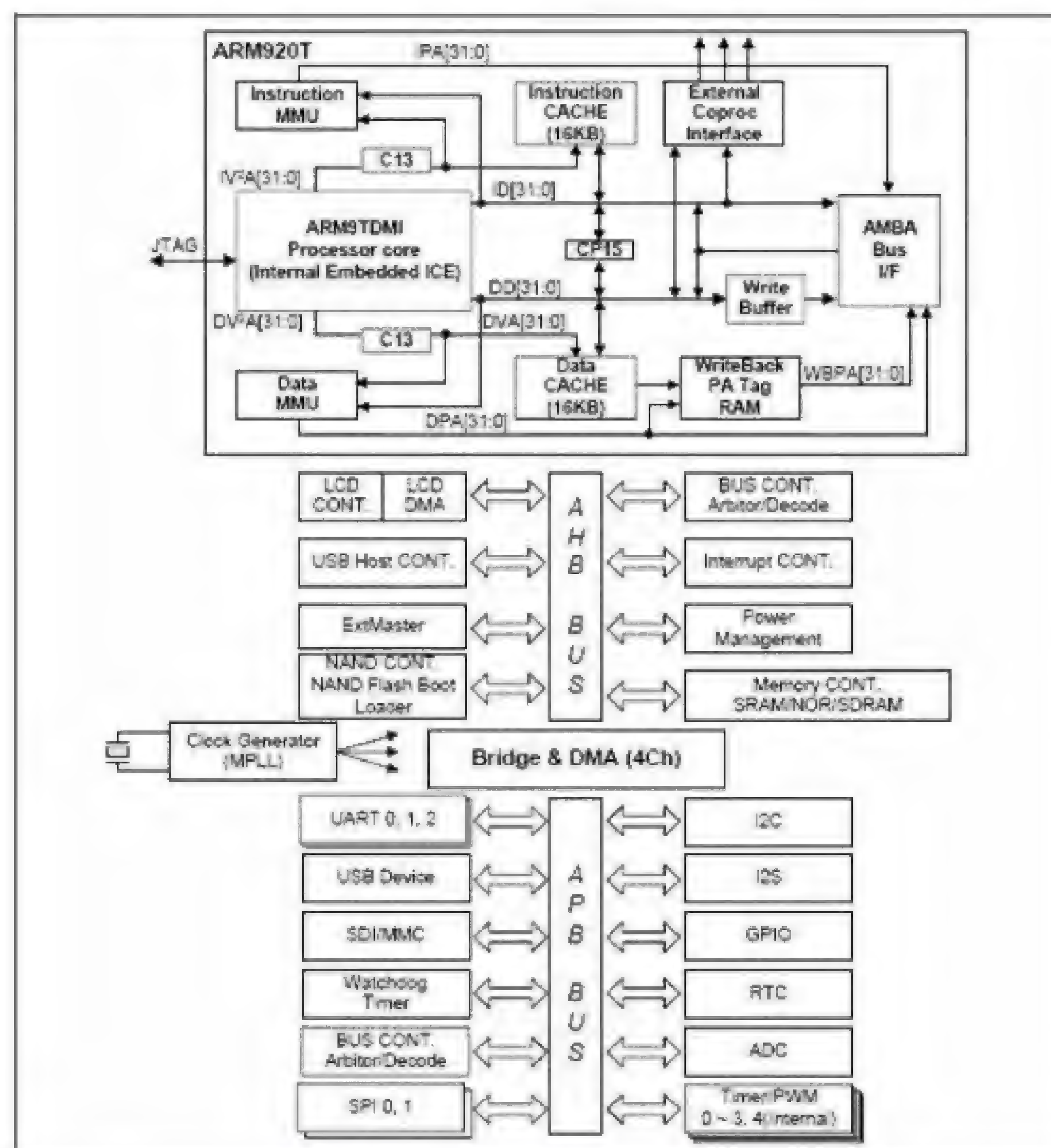


图 18.2 S3C2410X 内部结构图

S3C2410X 的显著特性是它的 CPU 核心是一个由 ARM 公司设计的 16/32 位 ARM920T RISC 处理器。ARM92OT 实现了 MMU，amba bus 和 harvard 高速缓冲体系结构。这一结构具有独特的 16KB 指令 cache 和 16KB 数据 cache，每个都是由 8 字长的行构成。通过提供一系列完整的系统外围设备，S3C2410X 大大减少了整个系统的成本，消除了为系统配置额外器件的需要，它集成了以下片上功能：

- 1.8/2.0V 内核供电，3.3V 存储器供电，3.3V 外部 I/O 供电。
- 具有 16KB 的 i-cache 和 16KB 的 d-cache/MMU。
- 外部存储控制器(SDRAM 控制和片选逻辑)。
- LCD 控制器(最大支持 4K 色 STN 和 256K 色 TFT)提供 1 通道 LCD 专用 DMA。
- 4 通道 DMA，并有外部请求引脚。
- 3 通道 UART(IrDA1.0，16 字节 TxFIFO 和 16 字节 RxFIFo)/2 通道 SPI。
- 1 通道多主 IIC-BUS/1 通道 IIS-BUS 控制器。
- 兼容 SD 主接口协议 1.0 版和 MMC 协议 2.11 兼容版。
- 2 端口 USB 主机/1 端口 USB 设备(1.1 版)。
- 4 通道 PWM 定时器和 1 通道内部定时器。
- 看门狗定时器。
- 117 个通用 I/O 口和 24 通道外部中断源。
- 功耗控制模式：具有普通、慢速、空闲和掉电模式。
- 8 通道 10bit ADC 和触摸屏接口。
- 具有日历功能的 RTC。
- 具有 PLL 片上时钟发生器。

18.3.2 交叉编译环境的建立

如果一个系统可以在不同的硬件平台上运行，那么称这个系统是可移植的。Linux 操作系统可以通过移植，运行在 ARM、PowerPC、M68k 等多种硬件平台上。

Linux 向其他的架构移植的时候需要有编译器的支持，而只有像 Gcc 这种强大的编译工具支持了这种架构后，Linux 才能被编译成可以在这种硬件架构上可执行的二进制代码。

通常情况下，由于嵌入式系统的资源有限，因而将包含 Gcc 编译器的 GNU 工具链安装于 PC 主机，而目标嵌入式系统并没有编译器，只负责存放编译好的可执行代码。这样的运行在主机上，但是生成的可执行文件只能在目标主机上运行的 GNU 开发工具叫作交叉工具链。

交叉工具链由一套用于编译、汇编和链接内核及应用程序的组件组成，这些组件包括编译器、调试器、链接和一些辅助工具，它们的说明如表 18.1 所示。

表 18.1 GNU 工具链组件说明

组　件	说　明
Binutils	用于操作二进制文件的实用程序集合，它们包括诸如反编译器 objdump、汇编器 as、连接器 ld 等
Gcc	编译器。可以实现交叉编译，即在宿主机上开发编译目标硬件上可运行的二进制文件
Glibc	所有的用户应用程序都将链接到 C 库。避免使用任何 C 库函数的内核和其他应用程序，可以在没有该库的情况下进行编译
Gdb	调试器，可以使用多种交叉调试方式，如背景调试 gdb_bdm 和网络调试器 gdbserver

说 明

根据笔者的经验，交叉工具链的构建过程对内存和硬盘的需求是巨大的。如果没有足够的内存和硬盘空间，那么在构建阶段由于相关性、配置或头文件设置等问题会突然冒出许多问题。因此，更值得推荐的做法是从因特网上下载预编译好的交叉编译工具链(本项目使用了 cross-2.95.3.tar.bz2)，然后安装于/usr/local/arm 目录下面。安装成功后，就可以看到交叉编译器 arm-linux-gcc 在/2.95.3/bin/这个目录下面。在这个目录下面，还会看到许多可能用到的二进制工具，例如 arm-linux-ar、arm-linux-ld、arm-linux-as、arm-linux-nm 等。

应用交叉工具链进行嵌入式应用程序的开发需要建立交叉编译环境。图 18.3 说明了交叉编译环境的建立。

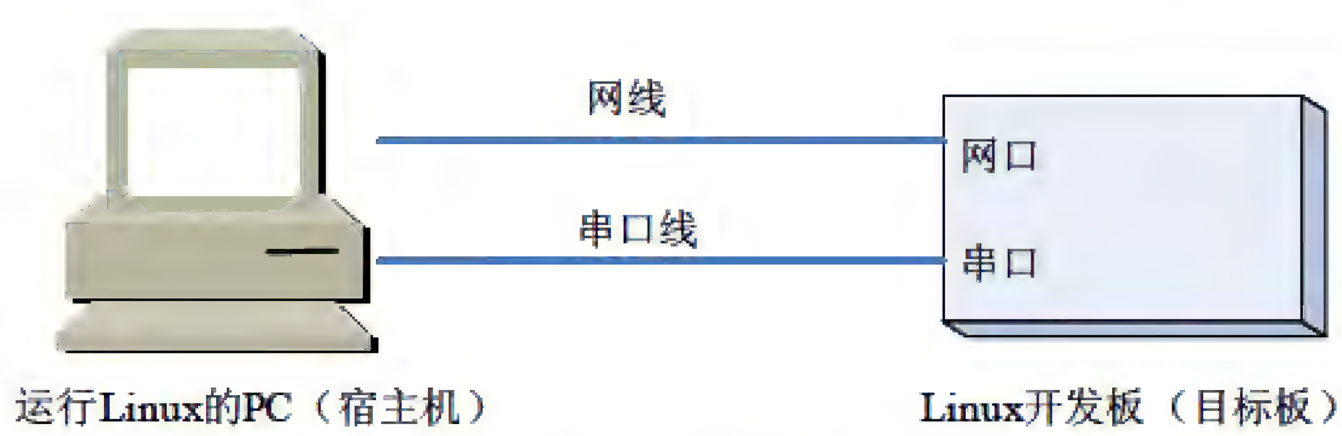


图 18.3 交叉编译环境的建立

通常情况下，我们将装有 Linux 操作系统的 Intel x86 架构的 PC 主机称为宿主机，用来开发和编译应用程序，然后通过网线或串口线将编译生成的可执行文件下载到目标板上(通常，网线的下载速度要比串口线快)，并在目标板上运行这些应用程序。串口线的另外一个重要功能是

使用终端(比如 Windows 下的超级终端、Linux 下的 minicom 等)打印目标板的运行信息，并对目标板进行相应的操作和控制。

建立交叉编译环境的首要任务是在 PC 机上安装进行嵌入式开发的操作系统，本书中的源代码所使用的开发环境都是 Red Hat Linux 9.0。

通常来说进行嵌入式开发都需要一个交叉编译环境，概括地讲，交叉编译就是在一个平台上生成另一个平台上的可执行代码。对于交叉开发方式，一方面开发者可以在熟悉的主机开发环境下进行程序开发；另一方面又可以真实地在目标板系统上运行调试程序，可以避免受到目标板硬件的限制。交叉编译的开发方式贯穿了嵌入式 Linux 系统开发的全过程。在本章所设计的课题中，笔者所使用的交叉编译过程模型如图 18.4 所示。

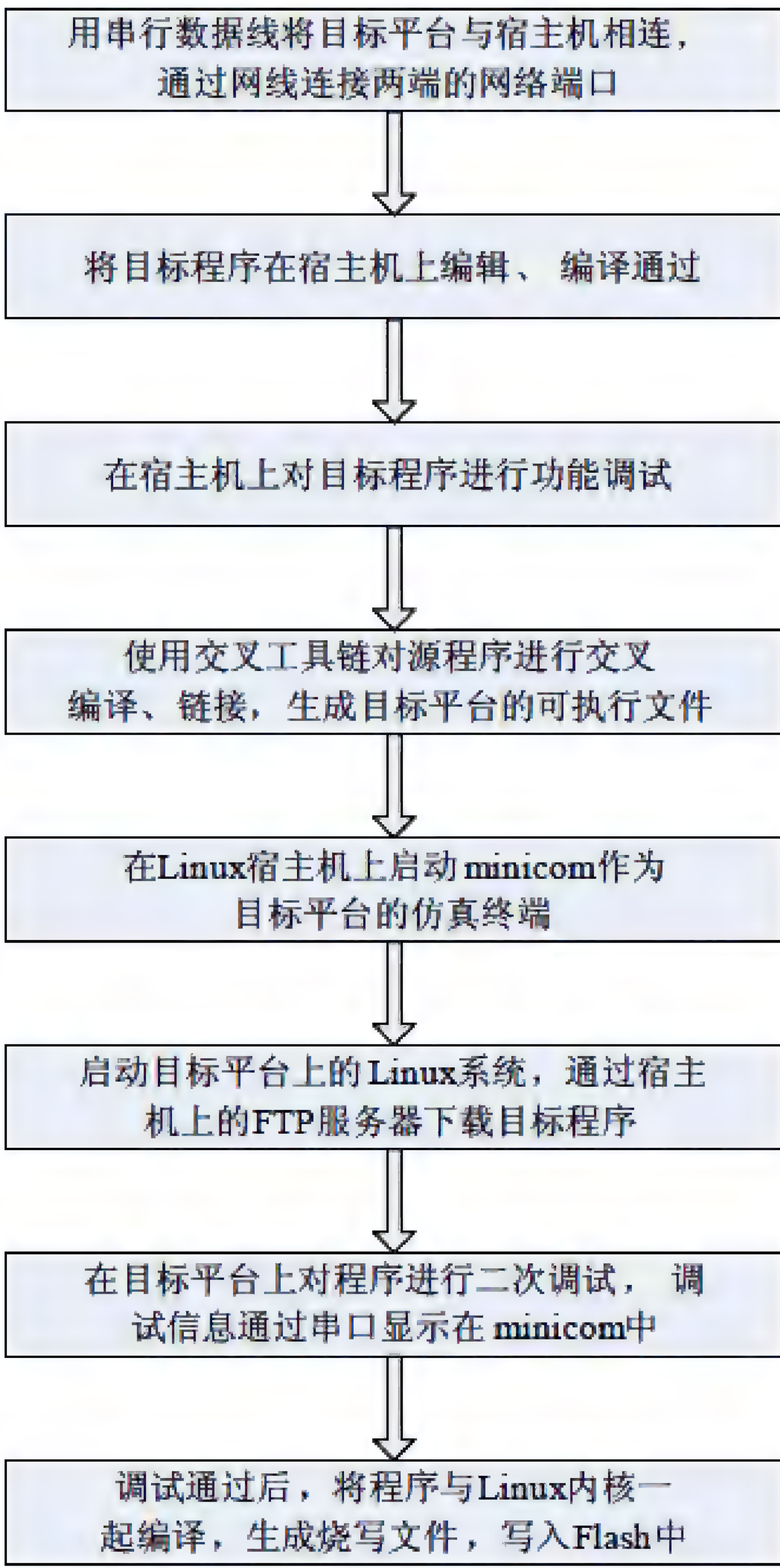


图 18.4　嵌入式开发的交叉编译

18.4 远程交互平台的设计

在搭建好开发平台的基础上，本节将围绕嵌入式家庭网关的应用软件设计进行一些研究和探讨。首先讨论开发模式的选择，通过对比分析采用 B/S 结构，选取 Boa 作为嵌入式 Web 服务器，并介绍通用网关接口 CGI 的概念及其工作原理。

18.4.1 应用软件的开发模式

C/S 结构和 B/S 结构是当今应用软件开发模式技术架构的两大主流技术。C/S 结构是美国

Borland 公司最早研发的，B/S 是美国微软公司研发的。

1. C/S 结构开发模式

C/S(Client/Server)模式即客户机/服务器模式，它具有两层结构，即客户机和数据库服务器。在前面几章的实例设计中，我们都采用了基于 C/S 模式的设计，相信读者对它也有了一定的理解。作为对比，这里重新说明。C/S 结构开发模式如图 18.5 所示。

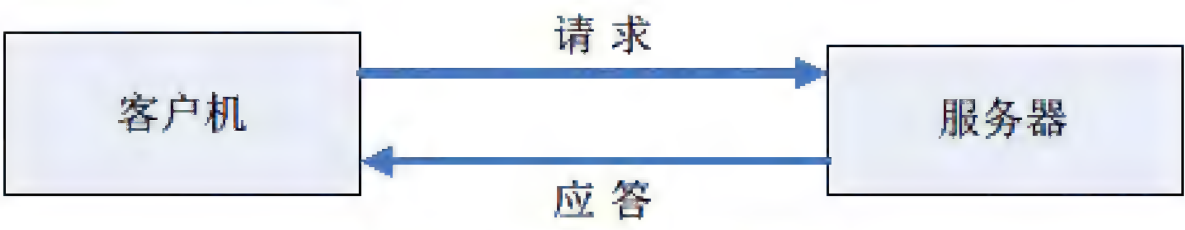


图 18.5　C/S 开发模式

在 C/S 开发模式下，由于客户端实现与服务器的直接相连，因此降低了网络通信量，事务处理速度快，交互性较强，有利于处理大量数据。它需要客户端安装专用的客户端软件，因此，客户端操作界面设计可以个性化，满足客户个性化的操作要求，具有直观、简单、方便的特点。

但是 C/S 模式开发是有针对性的，在特定的应用中，无论是客户端还是服务器端，都需要特定的软件支持。由于没能提供用户真正期望的开发环境，C/S 模式的软件需要针对不同的操作系统开发不同版本的软件，加之产品的更新换代十分快，在维护、系统升级方面都有很大的麻烦。另外，它的开发成本较高。

2. B/S 结构开发模式

B/S(Browser/Server)模式即浏览器/服务器模式，它由客户端浏览器、Web 服务器和数据库服务器 3 个部分组成，结构如图 18.6 所示。



图 18.6　B/S 开发模式

B/S 开发模式是随着 Internet 技术的兴起，对 C/S 模式的一种变化或者改进的模式。在这种模式下，用户工作界面是通过浏览器来实现，极少部分事务逻辑在前端实现，主要事务逻辑在服务器端实现。它具有分布性的特点，可以随时进行数据处理。在维护方面，只需要修改服务器端的网页，即可以实现所有用户的同步更新。同时，它实现了跨平台的系统集成服务，提供了异种机、异种网、异种应用服务的互联。

因此，鉴于以上分析，嵌入式家庭网关交互平台的软件设计采用 B/S 结构开发模式。

18.4.2　嵌入式 Web 服务器

采用 B/S 模式，首要的一个核心问题就是关于嵌入式 Web 服务器的选取和架设。先来介绍一下关于嵌入式 Web 服务器的有关理论。

1. 嵌入式 Web 服务器概述

随着网络技术的不断发展，使得各种嵌入式设备进行网络互联变为可能。同时，随着嵌入式操作系统，TCP/IP 协议栈在嵌入式设备中的广泛应用，使得这些设备的功能更为强大，结构更为复杂，互联性更为普遍。因此对这些众多的联网设备的访问、控制和管理也变得更加复杂

和重要。嵌入式 Web 技术为这种管理和控制提供了简单方便的实现方式。通过一个标准浏览器作为客户端，一个在设备中运行的嵌入式 Web 服务器作为服务端，就可以对远程的嵌入式设备和系统进行管理和配置了。

与传统的 Web 应用相比，嵌入式 Web 服务器(Embedded Web Server，EWS)简化了系统结构。由于有了标准的接口形式和基于 TCP/IP 的通信协议，内嵌于设备的 Web 服务器可以向任何接入它所在网络的合法用户提供统一的基于浏览器方式的操作和控制界面，浏览器成了设备的前端控制板。一般而言，嵌入式 Web 服务器应具有如下的特征：

- 占用较少的代码空间。
- 能够支持动态网页的生成。
- 可以与仪器方便的集成。

嵌入式 Web 服务器的软件系统包括 5 个部分，分别为 HTTP 引擎、虚拟文件系统、配置模块、安全模块、应用序接口模块。其组成如图 18.7 所示。

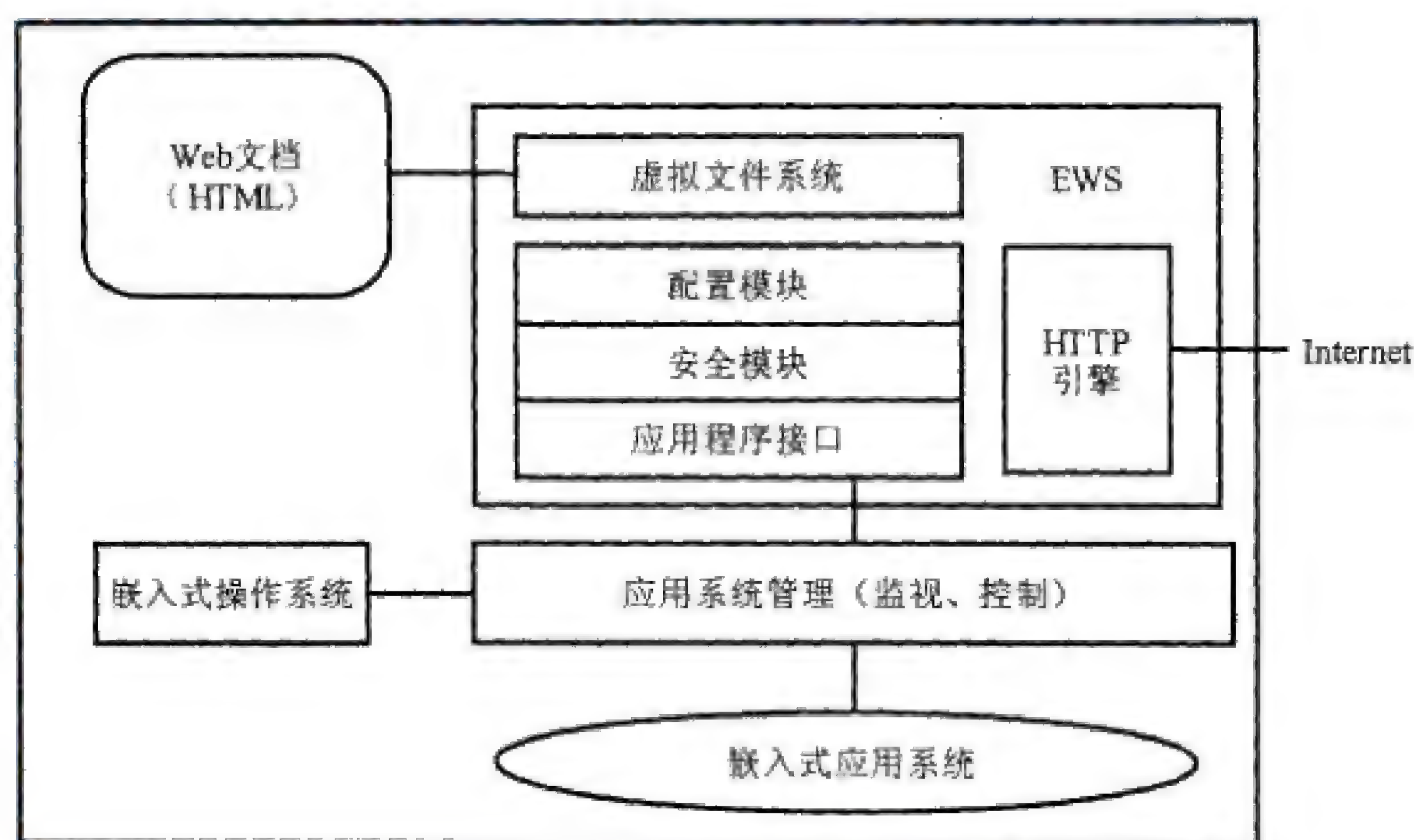


图 18.7 EWS 的软件系统组成

其中 HTTP 引擎负责响应用户的请求，通过虚拟文件系统访问静态数据信息，以及通过应用程序接口得到动态数据信息。

虚拟文件系统为 EWS 提供虚拟文件服务，虚拟文件系统使用数据结构存储文件大小、修改时间等信息。对于存储 HTML 文件需要的动态信息，可建立数据结构保存脚本的指针和脚本所调用函数的名称。通过虚拟文件系统将 Java、GIF、PDF、HTML，以及文本等文件形式编译为 Web 服务器认可的代码，而独立于具体的文件系统。

配置模块使系统管理员可以从任何一台标准的 Web 浏览器上设置 EWS 参数，在系统启动中定义的配置环境变量包括并发连接数、Socket 端口、主机名称、根文件路径、默认初始文件及非活动超时和时区等。

配置模块对标准浏览器的开放使得安全问题更加重要，尤其是对网络设备的配置和控制信息的访问成为安全保护的重点。安全模块通过在服务器上定义安全域和对每个安全域定义的用户名、密码实现对敏感信息的保护。还可以对请求数据采取加密措施实现安全保护功能。

应用程序接口模块实现和嵌入式应用系统的数据交换，在 EWS 中，应用程序接口与嵌入式操作系统通信，实现对嵌入系统的配置、监视和控制，是 EWS 软件系统的核心。应用程序接口模块常见的有 CGI(Common Gateway Interface)、SSI(Server Side Include)和 HCPA(HTML-to-C Preprocessor Approach)等 3 种形式。

随着 Web 技术的发展，嵌入式 Web 服务器技术在远程监控和生产过程控制系统中得到了广泛的应用。目前国外的相关研究有很多，如 Pharlap 公司的 MicroWeb、AgranatSystems 公司的 EmWeb、EmWare 公司的 emMicro、Allegro 公司的 RomPager、WindRiver 公司的 Wind，还有 Boa、Enea 等，国内也有(如 Webit)。并且这其中有不少开源的软件，因此在本章的课题开发过程中没有必要再去开发一个新的嵌入式 Web 服务器软件，而只需要根据自己课题的用途、硬件进行裁剪，移植即可，然后把大量的精力放在服务器端的应用软件的开发上来。本课题中

正是基于这种思路设计了家庭网关中的嵌入式 Web 服务器。

2. 嵌入式 Web 服务器 Boa

在 Linux 下，有很多功能强大的 Web 服务器，主要包括 Apache(第 16 章已向读者介绍)、httpd、thttpd 和 Boa。httpd 是最简单的一个 Web 服务器，它的功能最弱，不支持认证，不支持 CGI。thttpd 和 Boa 都支持认证、CGI 等，功能都比较全。Boa 是一款单任务的小型 HTTP 服务器，源代码开放、性能优秀，特别适合应用在嵌入式系统中。在本章的设计课题中选用 Boa 作为嵌入式 Web 服务器。

嵌入式 Web 服务器 Boa 和普通 Web 服务器一样，能够完成接收客户端请求、分析请求、响应请求、向客户端返回请求结果等任务。它的工作过程主要包括：

(1) 完成 Web 服务器的初始化工作，如创建环境变量、创建 TCP 套接字、绑定端口、开始侦听、进入循环结构，以及等待接收客户浏览器的连接请求。

(2) 当有客户端连接请求时，Web 服务器负责接收客户端请求，并保存相关请求信息。

(3) 在接收到客户端的连接请求之后，分析客户端请求，解析出请求的方法、URL 目标、可选的查询信息及表单信息，同时根据请求做出相应的处理。

(4) Web 服务器完成相应处理后，向客户端浏览器发送响应信息，关闭与客户机的 TCP 连接。

Boa 的功能实现是通过建立连接、绑定端口、进行侦听、请求处理等来实现的。它的整个工作流程如图 18.8 所示。

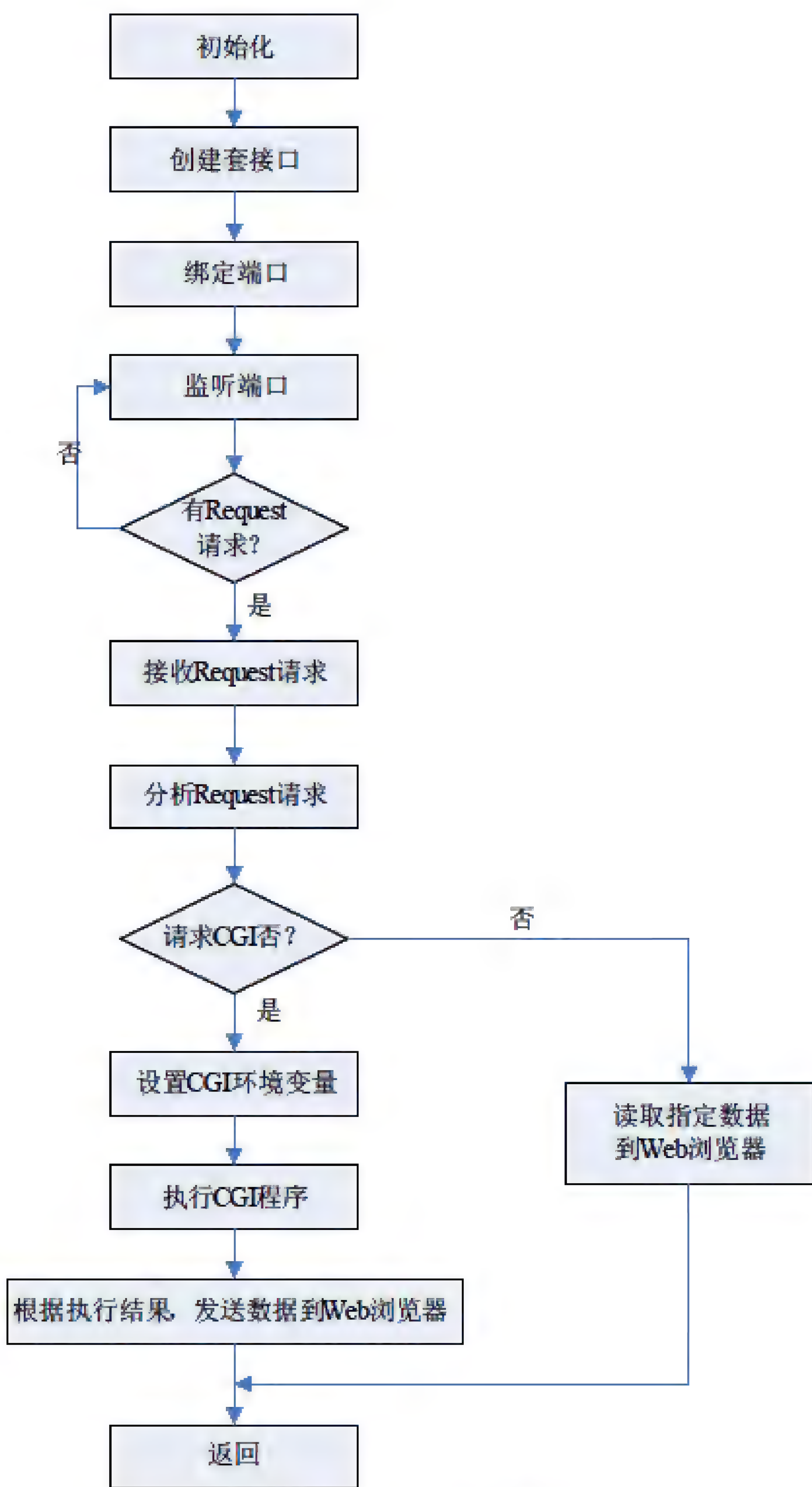


图 18.8　Boa 工作流程图

目前，在 Linux 操作系统中已经包含 Boa 的源代码，当然也可以从互联网(http://www.Boa.org)下载。在 Linux 下实现 Boa，只需要对 Boa 做一些简单的配置和修改，并在内核编译的过程中选择加载即可。关于 Boa 的配置、编译与移植过程，读者可查阅其他相关资料，鉴于篇幅，在此不做详细说明。事实上，目前市场上的很多嵌入式开发套件平台中都预装有嵌入式 Web 服务器 Boa，是可以直接使用的。

18.4.3 通用网关接口 CGI

读者应注意到，在前面的内容中已多次提到了 CGI。通用网关接口 CGI(Common Gateway Interface)是一种服务器与浏览器信息交换的标准接口。在物理上，CGI 是一段程序，它运行在服务器上，提供客户端 HTML 页面接口，完成 HTML 无法做到的交互功能。CGI 建立在客户机/服务器机制上，为外部扩展应用程序与 Web 服务器交互提供了一个标准接口。按照 CGI 标准编写的外部扩展应用程序可以处理客户端输入的工作数据，完成客户端与服务器的交互操作。如图 18.9 所示为使用 CGI 时，Web 浏览器和 HTTP 服务器之间的数据传输过程示意图。

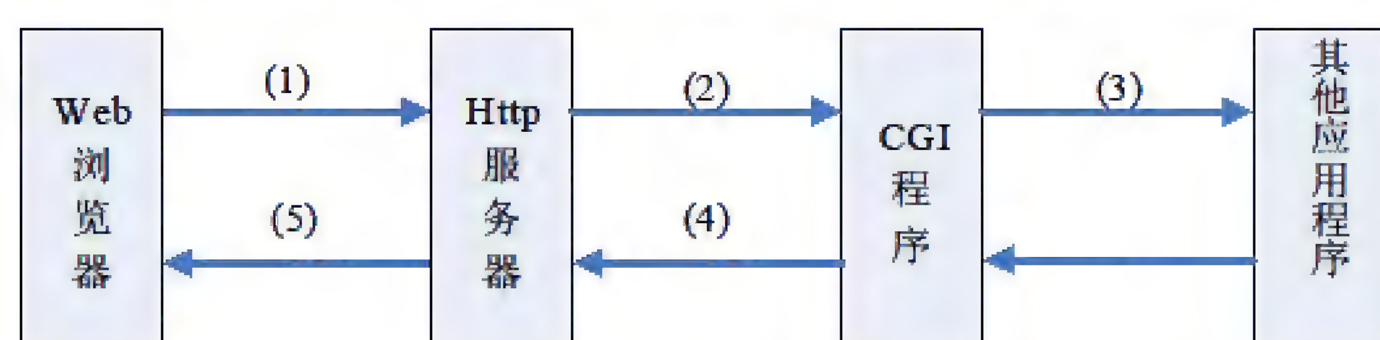


图 18.9　基于 CGI 的 B/S 数据传输示意图

其数据交互的具体步骤为：

(1) 浏览器向 HTTP 服务器发送请求，即输入标准的统一资源定位符 URL，该请求包括服务的类型、提供该服务的主机域名、CGI 程序名及用户数据等用户请求信息。

(2) HTTP 服务器解析该请求，当请求的是一个 CGI 程序时，HTTP 服务器将调用该 CGI 程序。

(3) CGI 程序解析用户输入信息并调用其他应用程序，这些信息通过环境变量、命令行参数或标准输入流传递给 CGI 程序。

(4) 如果 CGI 程序的处理结果需要返回给客户浏览器，则 CGI 程序必须为这些输出数据附加一个 HTTP 服务器可以理解的数据头。

(5) 服务器读取由 CGI 程序返回的数据，根据数据头信息，决定处理方式。如果数据头是“Content-type”类型，服务器将把数据送给客户机，由浏览器负责返回数据的处理和显示输出。最后，一旦浏览器收到所有的返回数据，即关闭与服务器的连接。

CGI 程序的调用方法有两种，一种是直接在浏览器 URL 中输入 CGI 程序调用，另外一种是在 HTML 程序中通过 FORM 表单调用。如果用户的 FORM 表单数据是 GET 方式，将通过环境变量 QUERY_STRING 传给 CGI 程序；如果是 POST 方式，将通过标准输入(stdin)传给 CGI 程序。主要的 CGI 环境变量有：

- GATEWAY-INTERFACE：CGI 程序所使用的 CGI 标准接口的版本号。
- REQUEST-METHOD：HTTP 请求方法，用以决定是通过 stdin 还是通过环境变量 QUERY-STRING 来获取客户端传输数据。
- QUERY-STRING：用于保存 CGI 程序 URL 中“?”之后的数据。

- CONTENT-LENGTH：表示客户端传输数据的字节数。
- CONTENT-TYPE：表示客户端传输数据的数据编码类型。

对于使用了属性“METHOD=GET”的表单(METHOD 属性的默认值为 GET)，CGI 定义为当表单被发送到服务器端后，表单中的数据被保存在服务器上一个叫作 QUERY_STRING 的环境变量中。这种表单的处理方法是读取环境变量 QUERY_STRING。在 C 语言中，可以用库函数 getenv 来把环境变量的值作为一个字符串来存取，然后再进行具体细节的处理。

对于使用了属性“METHOD=POST”的表单，CGI 定义为当表单被发送到服务器端后，表单中的数据被送到 CGI 程序的标准输入(在 C 语言中是 stdin)，而被传送的长度被放在环境变量 CONTENT_LENGTH 中。这种表单的处理方法是在标准输入中读入 CONTENT_LENGTH 长度的字符串。对于环境变量 CONTENT_LENGTH 的读取和处理方法，与环境变量 QUERY_STRING 类似。从标准输出读入数据要注意一些细节的地方，不能读多于 CONTENT_LENGTH 长度的字符，否则会造成严重的后果。

18.5 Linux 下软件模块的具体实现

课题中，嵌入式家庭网关应用系统实现的功能为：用户可以在远离家庭的地方通过 PC 机或者各种手持设备的 Web 浏览器对智能设备进行访问和控制。嵌入式家庭网关可以实现设备的监控、远距离数据的采集，比如对于空调、电冰箱等智能家电的控制，对于水表、电表、气表等实现远程自动抄表功能等。嵌入式家庭网关完成用户应用层协议的转换，将用户指令解释成智能设备能识别的控制命令。

由于试验条件的限制，课题中采用 2 片 MSC-51 单片机系统来分别模拟 RS485 接口中央空调和 RS485 接口智能水表，以此来达到验证嵌入式家庭网关应用系统的功能。

软件模块可以划分为以下几个部分：登录验证模块、中央空调监控模块、智能水表数据采集模块。后两个模块将对 RS485 总线进行读写，都调用网关的串口通信模块。系统的逻辑框图如图 18.10 所示。省略号表示这里可以有多个受控模块，为简化本章课题的设计内容，这里只设计了两个模块。事实上，各个模块的工作及流程是大同小异的。

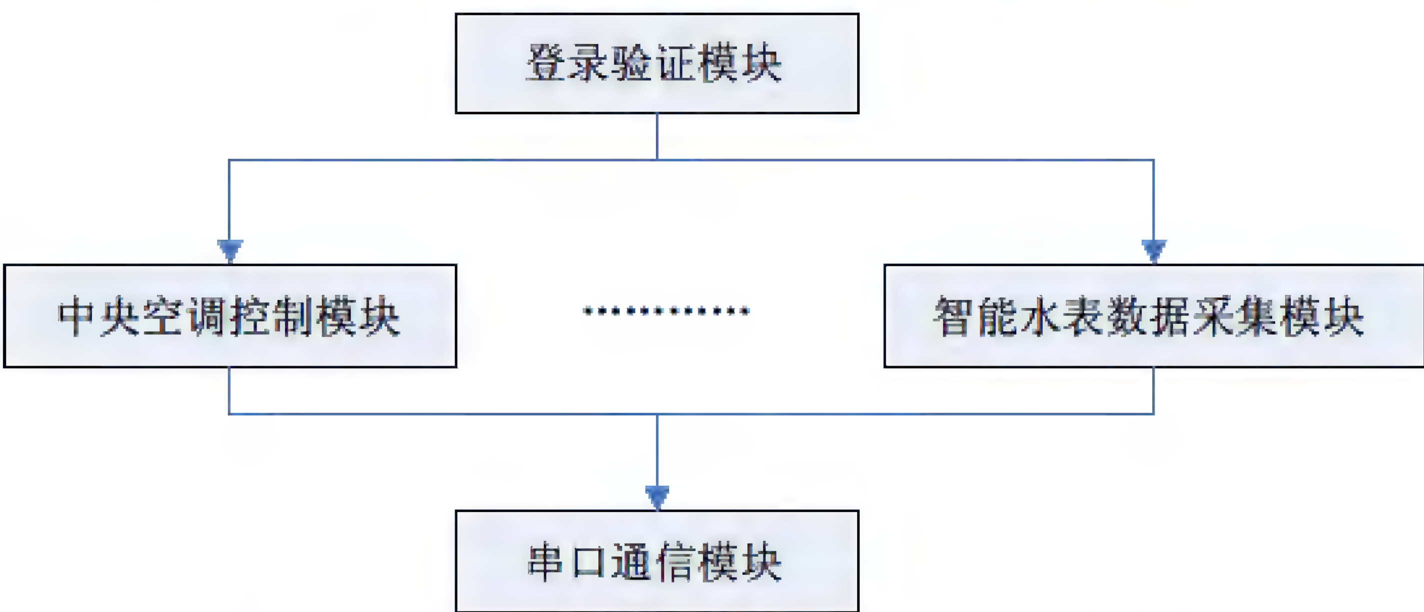


图 18.10　嵌入式家庭网关应用系统逻辑框图

下面向读者分别介绍 Linux 下各个软件模块的具体实现流程，有关硬件部分的原理与设计，已超出本书的范围，这里只进行简单的讲解。读者可查阅其他资料，也可以参考本书附带光盘中的源代码。

18.5.1 登录验证模块

登录验证模块用来验证用户的身份，这在一定程度上保证了系统的安全性。用户通过填写远程 Web 界面的表单数据，并将表单提交至内置在家庭网关中的嵌入式 Web 服务器 Boa，服务器 Boa 便会调用登录验证模块的 CGI 程序 login.cgi。

login.cgi 程序解析远端传送来的表单数据，包括用户名和密码信息，并将解析的结果与数据库文件 password 中的数据对比，相同则说明用户名和密码正确，生成家庭网关应用系统的初始网页，等待远程用户的操作；若不相同，则说明用户名和密码有误，则生成错误信息网页，提示用户重新输入。login.cgi 的工作流程如图 18.11 所示。

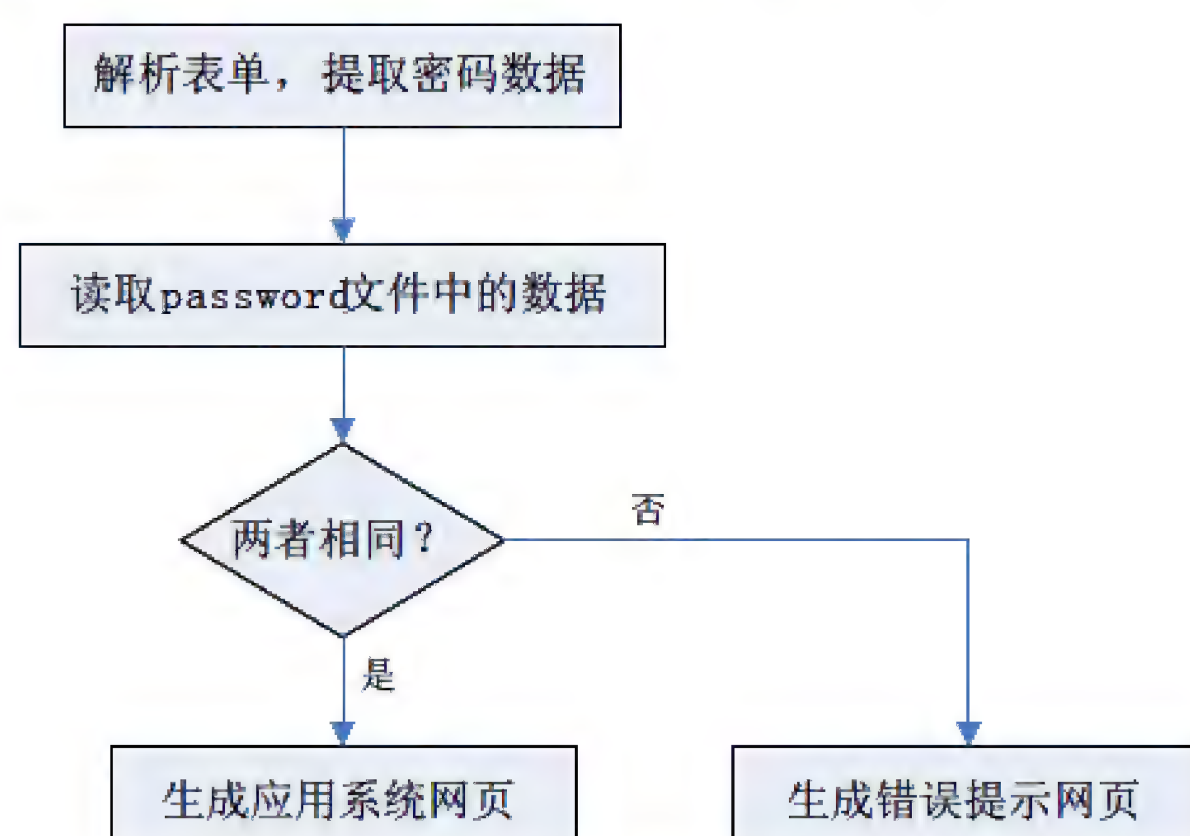


图 18.11 登录验证模块流程图

18.5.2 串口通信模块

嵌入式家庭网关通过 RS485 总线来访问和控制具有 RS485 接口的智能设备(如模拟的中央空调和智能水表)，采用半双工通信方式，嵌入式家庭网关作为主节点(主控)，其他所有的 RS485 接口的智能设备都是从节点(受控)。每次都是由嵌入式家庭网关发出一条信息帧，其中包括要访问和控制的智能设备的地址，和相应操作的控制命令。各个从节点事先都定义好自己的地址，当 485 总线有数据时，将全部的帧信息接收下来，然后和自己的地址进行比较，当嵌入式家庭网关发送的帧中包括自己的地址时，此从节点则对此帧进行应答处理，其他节点则忽略此帧，不进行任何处理。

家庭网关与所有智能设备的串口通信模块采用 Modbus 协议。Modbus 协议是应用于电子控制设备上的一种通用语言。通过此协议，控制器相互之间、控制器经由网络(例如以太网)或各种数据总线与其他设备之间可以通信，它已经成为一种通用的工业标准。通过 Modbus 协议，不同厂商生产的控制设备可以连成工业网络，进行集中监控。

Modbus 协议定义了一个控制器能够识别的消息结构，而不管它们是经过何种网络或总线进行通信的。它描述了控制器请求访问其他设备的过程，和来自其他设备的回应信息，以及怎样侦测错误并记录。它制定了消息域的格局和内容的公共格式。

当在一个使用 Modbus 协议的网络上通信时，此协议决定了每个控制器都需要知道它们的设备地址，识别按地址发来的消息，决定要产生何种行动等。如果需要回应，控制器将生成反馈信息并用 Modbus 协议发出。Modbus 协议的数据帧格式如图 18.12 所示。

起始标记	从机地址	命令代码	数据个数	N字节数据	CRC校验和
2字节	1字节	1字节	1字节	N字节	2字节

图 18.12　Modbus 协议数据帧格式

数据帧中各个消息域的含义与内容如下：

- 起始标记：标记每一个数据帧的开始，2 字节内容为 0XEB 和 0X90。
- 从机地址：受控设备的地址，0X01～0X20。其中 0X00 地址留为广播模式访问。
- 命令代码：对于不同的设备，有不同的定义。
- 数据个数：指后接 *N* 字节数据的字节数 *N*，不宜太长。
- *N* 字节数据：对于 char 型数据，原样传输。对于 int 型、float 型、double 型数据，采用一种比较简单的方式，事先约定好整数部分和小数部分的位数，不足位补零，然后按位传输。
- 校验和：采用 CRC-16 对数据帧进行校验，校验多项式为 $X^{16}+X^{15}+X^{2}+1$。

18.5.3　中央空调控制模块

在中央空调控制模块中，远程用户通过浏览器，将表单信息提交给嵌入式 Web 服务器，服务器调用中央空调控制模块的 CGI 程序 aircondition.cgi。

接着，aircondition.cgi 程序分析表单的数据并形成 Modbus 协议的信息帧，然后调用串口通信模块，将用户的操作提交到具体的设备，即模拟中央空调控制模块的单片机系统。服务器等待单片机的处理及返回的信息帧，然后产生动态网页，返回给远端用户。其工作流程图如图 18.13 所示。

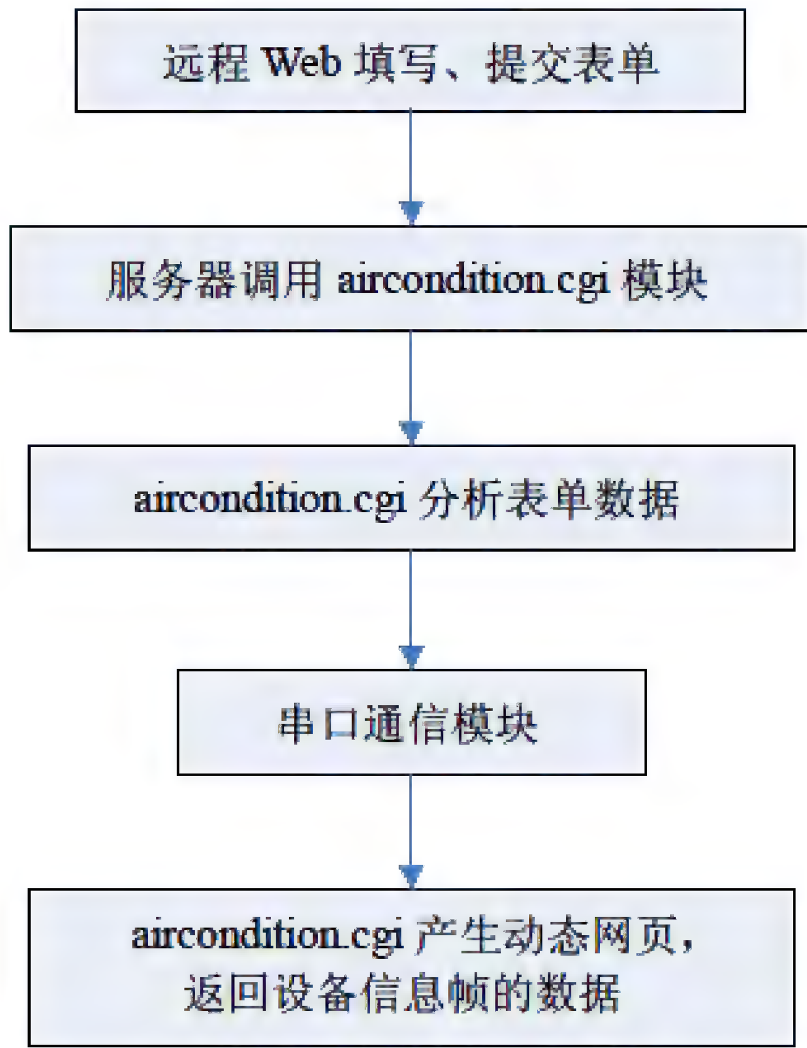


图 18.13　中央空调控制模块流程图

另外，需要定义一些中央空调控制模块的命令码，如表 18.2 所示。

表 18.2　中央空调控制模块命令码列表

命　令　码	含　　义
0X00	写入程序
0X10	调温

(续表)

命 令 码	含 义
0X11	定时
0X12	风向
0X13	风速
0X20	启动/关闭
……	……

在设计此模块的 Web 网页时，表单的界面如图 18.14 所示。表单的提交方式为 GET，所以将形成如下形式的 URL：

```
http://192.168.0.84/cgi-bin/aircondition.cgi?R1=V3&T2=20
```

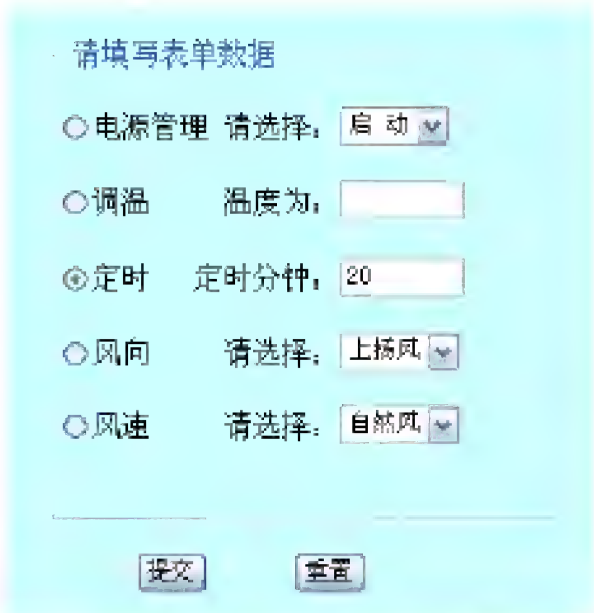


图 18.14 中央空调控制模块表单

在上面的字符串中，192.168.0.84 是嵌入式 Web 服务器(即 S3C2410 家庭网关平台)的 IP 地址，R1 是单选按钮的 name，V3 是单选按钮的 value；T2 是文本框的 name，20 是填入文本框的 value。上述表单中单选按钮的取值还有可能为 V1、V2、V4、V5，分别对应相应的操作，如图 18.14 所示。表单以 GET 方式提交时，将把 R1=V3&T2=20 等形式的数据存入环境变量 QUERY_STRING 中。该模块的 CGI 程序的实现方法如下(取自光盘的/src/chapter_18/cgic/aircondition.cgi 文件)：

```
#define ADDR 0x10
#define MAXLEN 40
void main()
{
char *data;        /*指向环境变量的数据*/
char command[3];   /*保存提交的表单中的命令项*/
char info[33];     /*保存提交的表单中的数据项*/
char frame[MAXLEN];     /*串口通信的信息帧*/
frame[0]=0XEB;     /*填写帧的起始标记字段*/
frame[1]=0X90;
frame[2]=ADDR;   /*填写帧的从机地址字段，即空调设备的地址*/
int len=3;           /*信息帧的长度*/
int*framelen=&len;
data=getenv("QUERY_STRING");
if(data==NULL)
{
printf("<p>错误!");
}
else
{
command[0]=data[3];
command[1]=data[4];
command[2]='\0';
```

```
info=substr(data,8,strlen(data)-8);     /*获取有效字符串数据*/
if(strcmp(command=='V1'))          /*启动/关闭*/
{
    …………          /*启动/关闭的相关操作*/
}
else if(strcmp(command=='V2'))     /*调温*/
{
    …………          /*调温的相关操作*/
}
else if(strcmp(command=='V3'))     /*定时*/
{
proc1(info);          /*对 info 进行一些处理*/
frame[3]=0X11;        /*填写帧的命令代码字段*/
frame[4]=strlen(info); /*填写帧的数据个数字段*/
strcat(frame,info);    /*填写帧的数据项字段*/
CRCverify(frame);      /*填写帧的 CRC 验证字段*/
len+=strlen(info)+4;
if(communicate(frame,framelen))     /*调用串口通信模块*/
{
…………     /*产生动态网页，返回设备数据*/
}
else
{
…………     /*关于出错处理的网页提交*/
}
}
else if(strcmp(command=='V4'))    /*风向*/
{
    …………        /*风向调节相关操作*/
}
else if(strcmp(command=='V5'))    /*风速*/
{
    …………       /*风速调节相关操作*
}
else
{
printf("<p>传输错误!");
}
}
}
```

这里仅仅描述了关于定时命令的信息帧的填充，串口通信及反馈动态网页部分的代码。其他命令操作方法的代码实现类似，在此就不多占篇幅，读者可参考光盘中的源代码。

模拟中央空调的单片机系统程序流程如图 18.15 所示。

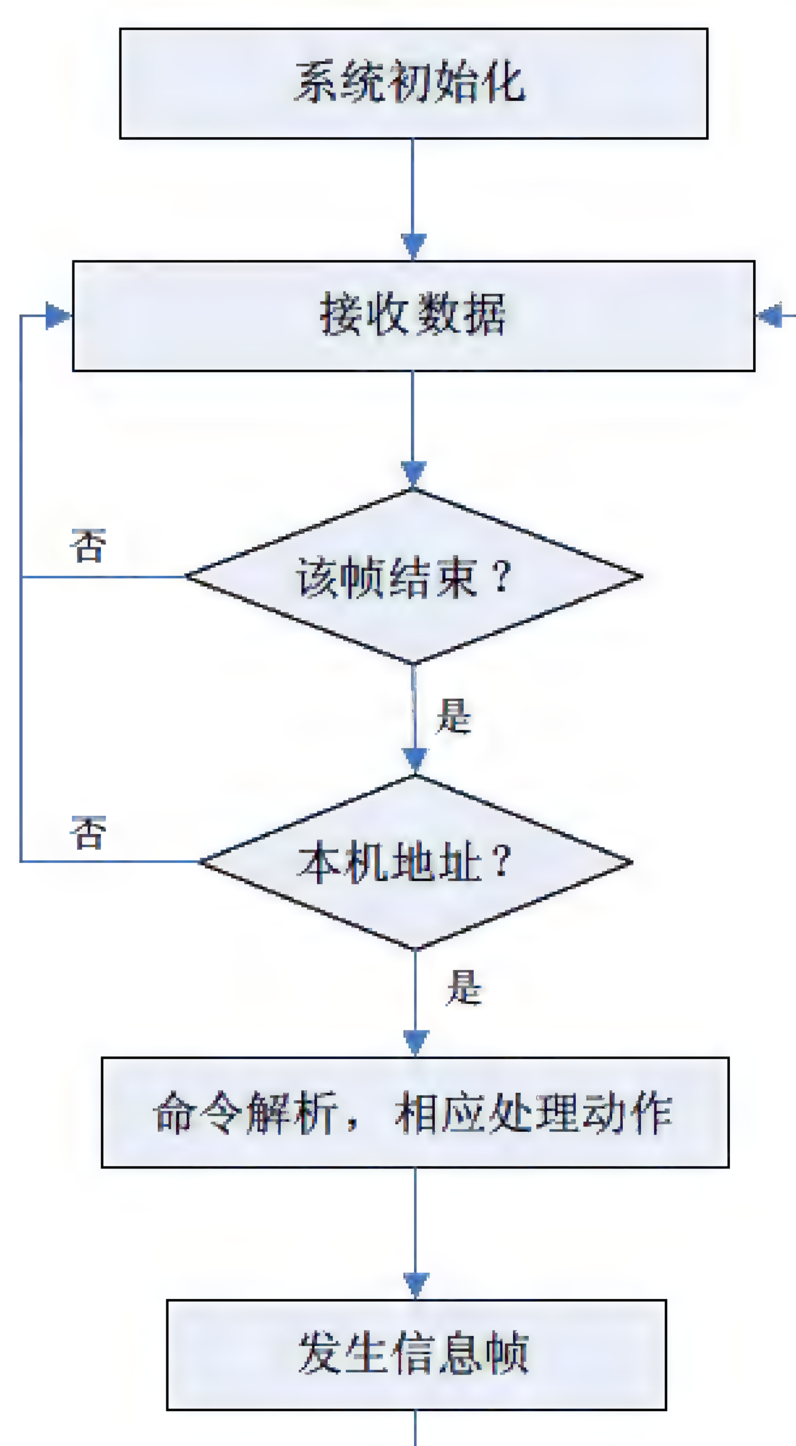


图 18.15　单片机系统程序流程图

需要注意的是，返回给主机的信息帧，其地址要改为主机地址或者本地地址，不能随机，否则会产生 485 总线的混乱。下面是对 MSC-51 单片机的编程，关于这一部分，读者可参考相应的单片机书籍。在本章课题中的实现如下(取自光盘的/src/chapter_18/ cgic/aircondition_msc51.c 文件)，鉴于篇幅有限，这里也只是给出了部分重要代码：

```
#define ADDR 0x10
#define MAXLEN 40
void main()
{
initsys();            /* msc51 初始化*/
char head[MAXLEN];
int*framelen;
while(1)
{
readframe (head, framelen);
if(head[4]==ADDR)
{
procframe (head, framelen);
sendframe (head,framelen);48
}
}
}

void initsys()
{
SCON=0x50;//SCON：串口模式 1
TMOD|=0x20;//TMOD：定时器 1，模式 2
```

```
PCON|=0x80;//SMOD=1;
TH1=0xFA;//波特率 9600 fosc=11.0592MHz
IE|=0x90;//使能串口中断
TR1=1;//定时器 1 运行
}

void readframe(char*head,int*framelen)    /*接收一帧数据*/
{
int i=4,j=-1,len=0;
char data;
char*p=head;
p1.7=1;       /*控制 485 接收数据*/
while(j)
{
data=readbyte();
*p++=data;
if(i==0)
{
j=data+3;
        }
i--; j--; len++;
}
*framelen=len;
}

char readbyte()    /*接收一个字节数据*/
{
char data;
while(!RI);
data=SBUF;
RI=0;
return data;
}

void procframe(char*head,int*framelen)   /*处理信息帧*/
{
char command=head[3];
if(command==0X20)
{
power(head,framelen);     /*启动/关闭模块*/
}
else if(command==0X10)
{
temperature(head,framelen);   /*启动调温模块*/
   }
else if(command==0X11)
{
timer(head,framelen);     /*启动定时模块*/
}
else if(command==0X12)
```

```
{
windway(head,framelen);     /*启动风向模块*/
    }
else if(command==0X13)
{
windspeed(head,framelen);   /*启动风速模块*/
}
else
    {
…………     /*其他处理模块*/
}
}

void sendframe(char*head,int*framelen)    /*发送一帧数据*/
{
int len=*framelen;
char*p=head;
p1.7=0;         /*控制 485 发送数据*/
while(len)
{
sendbyte(*p);
p++;
len--;
}
}

void sendbyte(char ch)      /*发送一个字节数据*/
{
SBUF=ch;
while(!TI);
TI=0;
}
```

这里的 power()、temperature()、timer()、windway()、windspeed()由于只是对中央空调的一个模拟，所以并没有具体意义的实现，仅仅是根据命令由单片机模拟的中央空调产生一个相应的处理过程，并返回一个应答的信息帧。

18.5.4 智能水表数据采集模块

智能水表数据采集模块的实现基本上和中央空调控制模块类似，所不同的是信息帧地址不同；具体的功能操作不同，带来相应的 CGI 处理模块也需略作改动。

这里需要说明的是，本章的重点并不在于智能水表的概念及其工作原理，课题的设计也并不关心智能水表是如何工作的。此模块的作用只是向读者展示嵌入式家庭网关的重要功能，鉴于可行性，智能水表数据采集模块也是使用 MSC-51 单片机来进行模拟的。嵌入式 Web 服务器(即家庭网关)发出控制命令及相应的操作参数，单片机解析这些命令和参数，执行相应的动作即可，比如点亮某一个管脚的二极管，以此来验证用户的操作是否被“智能水表”所接收。

模拟智能水表的单片机系统，其实现基本上也类似于模拟中央空调控制模块的单片机系统，仅仅是具体功能的处理有别。所以，关于智能水表数据采集模块的具体实现和关于模拟的

单片机系统的具体实现不进行类同介绍。为了降低系统的复杂度和试验成本，课题中也仅仅是用 2 个单片机系统来验证嵌入式家庭网关的具体实现。

18.5.5 试验结果

该应用系统基本上能模拟出嵌入式家庭网关的一些功能。嵌入式家庭网关通过 RS485 总线，能把 31 个具有 RS485 接口的智能设备连成内部测控网络，进行访问和控制。通过浏览器，用户可以在远方对家庭内部设备进行访问和控制，如对 RS485 接口中央空调控制器的操作，对 RS485 接口的智能水表的读取等。

首先，用户登录验证模块验证正确后，将产生系统的欢迎界面。此时单击网页左边导航栏中的“中央空调控制器”链接，便生成中央空调控制模块的表单页面，如图 18.16 所示。

比如，选中“定时”单选按钮，在“定时分钟”文本框中填入 20，即对空调定时 20 分钟，20 分钟后将自动关闭中央空调，单击“提交”按钮提交表单，返回的页面如图 18.17 所示。

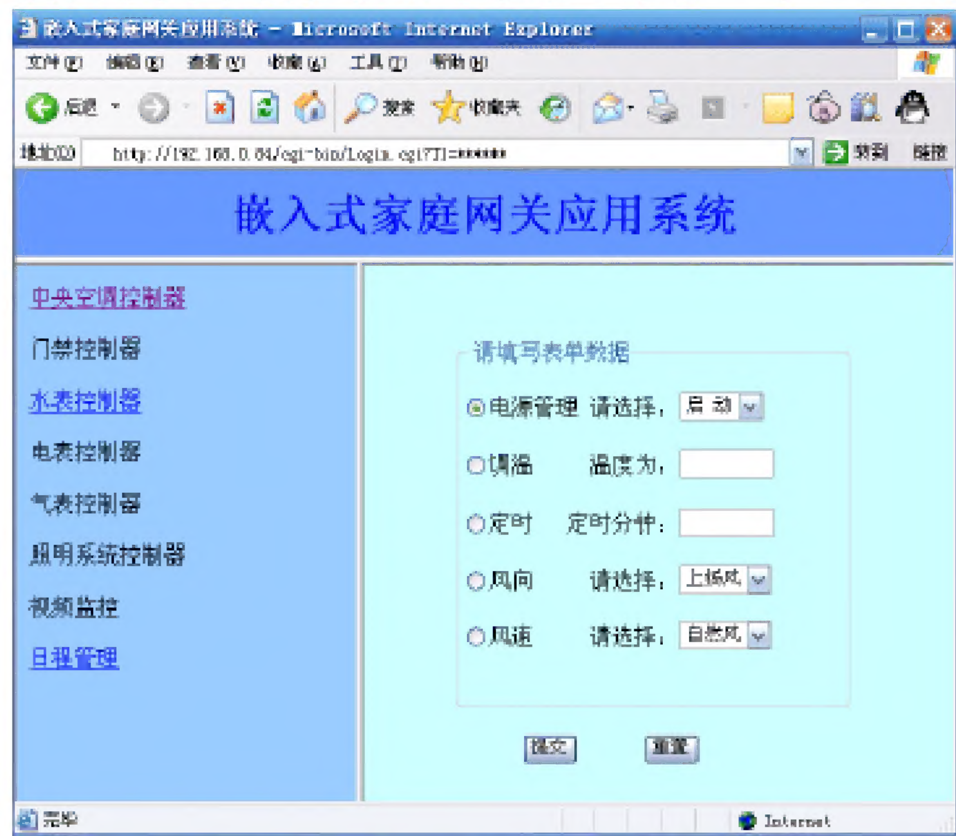


图 18.16 中央空调控制页面

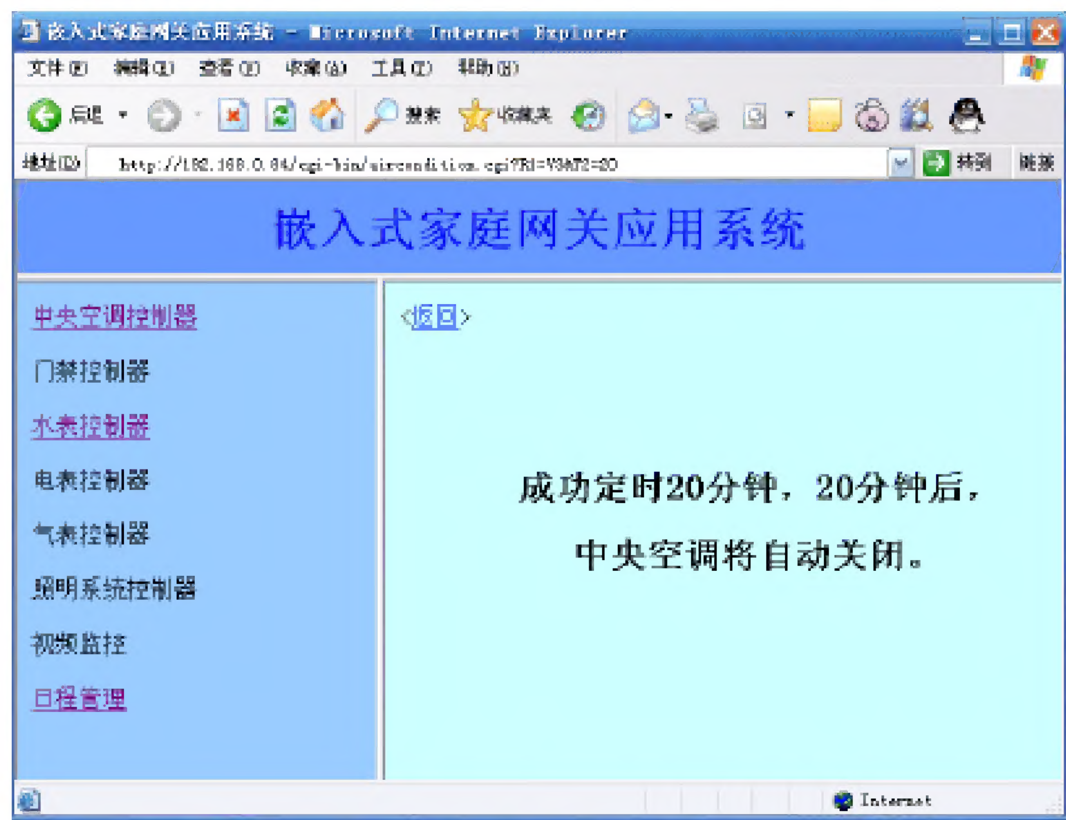


图 18.17 定时操作返回页面

此外，还可以选择“水表控制器”链接，远程读取智能水表的信息。鉴于篇幅有限，在此不一一列举。

最后需要说明一点，系统的测试是在局域网内进行的，由于试验条件，并没有测试于广域网。但在广域网范围内的原理类似，可以推测，有兴趣的读者可以亲自试一试。

18.6 本章小结

智能家庭网络技术近几年来已成为家电、通讯、自动化、房地产行业的一个热门话题，诸多跨国集团纷纷将目光投向该技术领域，并推出了各种技术解决方案，有力地推动了家庭网络数字化、信息化产业的迅猛发展。

家庭网关是智能家庭网络物理上与逻辑上的核心。家庭网关集电信、家电、IT 技术于一体，是家庭网络联络外部公网的重要桥梁，由于网关必须和家庭内部的应用技术以及外部公众网络应用业务配套，因而国外还没有成熟的家庭网关标准出台，我国的家庭网关标准也在研究中。因此可以说，嵌入式家庭网关必定会成为我国未来几十年的热门技术之一。

本章首先介绍了嵌入式技术和家庭网关的相关基本概念，接着着重介绍了嵌入式家庭网关远程交互操作平台的设计与实现。通过比较与分析，系统采用 B/S 结构的开发模式，嵌入式

Web 服务器选取 Boa，并介绍了 Boa 的工作原理。最后，以 MSC-51 单片机系统来模拟具体的智能家电设备(中央空调、智能水表)，并结合 CGI 技术，用 C 编写 CGI 程序，实现了动态的具体智能设备的访问和控制。